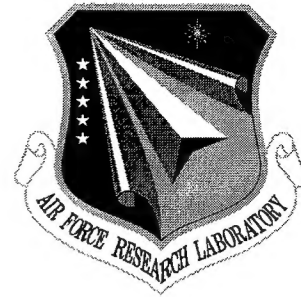AFRL-IF-RS-TR-2000-120
Final Technical Report
August 2000

SESAME: SCALABLE SYSTEMS SOFTWARE MEASUREMENT AND EVALUATION

University of California, Los Angeles

Sponsored by
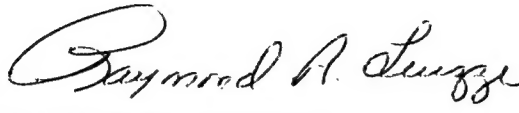Defense Advanced Research Projects Agency
DARPA Order No. B549

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

**AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK**

20001213 078

This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

AFRL-IF-RS-TR-2000-120 has been reviewed and is approved for publication.

APPROVED: RAYMOND A. LIUZZI
Project Engineer

FOR THE DIRECTOR: NORTHRUP FOWLER, III, Technical Advisor
Information Technology Division
Information Directorate

SESAME: SCALABLE SYSTEMS SOFTWARE
MEASUREMENT AND EVALUATION

Leonard Kleinvock
Rajive Bagrodia
Gerald Popek

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE August 2000	3. REPORT TYPE AND DATES COVERED Final Oct 94 - Feb 2000	

4. TITLE AND SUBTITLE

SESAME: SCALABLE SYSTEMS SOFTWARE MEASUREMENT AND EVALUATION

5. FUNDING NUMBERS

C - F30602-94-C-0273
PE - 62301E
PR - B549
TA - 01
WU - 01

6. AUTHOR(S)

Leonard Kleinrock, Rajive Bagrodia, and Gerald Popek

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

University of California, Los Angeles
Computer Science Department
405 Hilgard Avenue
Los Angeles CA 90024-1406

8. PERFORMING ORGANIZATION REPORT NUMBER

N/A

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Defense Advanced Research Projects Agency   AFRL/IFTD
3701 North Fairfax Drive   525 Brooks Road
Arlington VA 22203-1714   Rome NY 13441-4505

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

AFRL-IF-RS-TR-2000-120

11. SUPPLEMENTARY NOTES

AFRL Project Engineer: Raymond A. Liuzzi/IFTD/(315) 330-3577

12a. DISTRIBUTION AVAILABILITY STATEMENT

Approved for Public Release; Distribution Unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

The goal of the SESAME project has been to create a set of tools to study the performance of operating systems. This project developed a method for observing component behavior inside the operating system kernel in fully running computers, a method for modeling the effects of changes, and a method for sharing the results among researchers. These have been fully implemented and tested on a massively parallel processing system (MPP). The project has used this implementation to identify details of system operation that were both contrary to documentation, and difficult to establish from user-level observation. In the process, two especially important lessons have been learned. The first is that extracting data from an OS kernel is a very challenging task. Although placing "diagnostic code" into an OS seems simple enough, to be useful the method must handle the large volumes of data produced without degrading system performance, organizie it in a consistent analyzable way, and make the process flexible and extensible enough to allow arbitrary investigations. The method and programming described are substantially portable, and other teams will almost certainly find it much more practical to adapt them, than to undertake a three-year process of reinvention from scratch. The second important lesson is that such invasive, detailed tools are likely best used by inside members of OS development teams, or their quality assurance counterparts. In short, exposing such intimate system behavior can expose trade secrets. The measurement team was bound by a non-disclosure agreement with the vendor, and faced a constant dilemma between its obligations not to disclose protected intellectual property, and pressure from analysts for detailed information.

14. SUBJECT TERMS

Computers, software, database, architecture

15. NUMBER OF PAGES

108

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

Table of Contents

Executive Summary

Modern multi-threaded operating systems are extremely complicated internally, with many components operating simultaneously. Although the design of each component may be fully understood by its creators, its behavior in a running system, interacting with other components, will not be. This often results in unexplained performance problems: expected speeds are not approached or systems crash under seemingly reasonable loads, without any single part being obviously defective.

Understanding the performance of the interacting components in a running system is essential to designers attempting to improve performance, for programmers attempting to optimize applications, and for computer scientists exploring new approaches. But no truly adequate method of observing actual system events or predicting the effects of changes has been available. Instead of reliable measurements and informed models, we must make do with outdated documentation, engineering folklore, and "reasonable" guesswork.

The goal of the SESAME project has been to create a set of tools to overcome this uncertainty. We have developed a method for observing component behavior inside the operating system kernel in fully running computers, a method for modeling the effects of changes, and a method for sharing the results among researchers. These have been fully implemented and tested on a massively parallel processing system (MPP). We have used them to identify details of system operation that were both contrary to documentation, and difficult to establish from user-level observation.

In the process, we learned two especially important lessons. The first is that extracting data from an OS kernel is a very challenging task. Although placing "diagnostic code" into an OS seems simple enough, to be useful the method must handle the large volumes of data produced without degrading system performance, organize it in a consistent analyzable way, and make the process flexible and extensible enough to allow arbitrary investigations. Our method and programming are substantially portable, and other teams will almost certainly find it much more practical to adapt them, than to undertake a three-year process of reinvention from scratch.

The second important lesson is that such invasive, detailed tools are likely best used by inside members of OS development teams, or their quality assurance counterparts. In short, exposing such intimate system behavior can expose trade secrets. Our measurement team was bound by a non-disclosure agreement with the vendor, and faced a constant dilemma between its obligations not to disclose protected intellectual property, and pressure from our analysts for detailed information. Other investigators would do well to address this issue early.

This report provides details of the methods developed under the SESAME project, and demonstrates how those methods can be used to gain an understanding of system behavior.

Staff

DARPA

Program Manager, DARPA ITO	Gary Koob
Technical Contact, AF Research Lab	Joseph Cavano

University of California, Los Angeles

Repository Principal Investigator	Leonard Kleinrock
Programmers	Mark Strohm
	Saiyin Leung
Student Programmer	Hisajiro Yoshikawa
Assistant to the PI	Brenda Ramsey
Modeling Principal Investigator	Rajive Bagrodia
Senior Development Engineer	Ewa Deelman
Graduate Students	Stephen Docy
	Andy Kahn
	Thomas Phan
	Sundeep Prakash

Platinum *technology*, inc.

Measurement Principal Investigator	Gerald Popek
Managers	Karen Hackett
	David Gallon
Programmers	Roger Noe
	Jerry Toman
	Alice Wang
Technical Expert	Roman Zajcew

Time Line

October	1994	Project start
April	1995	Parallel program simulator operational
October	1997	First instrumentation
November	1997	Repository online
June	1999	Parallel program simulator complete
October	1998	Instrumentation complete
November	1998	Main experimentation start
February	1999	Main experimentation complete
May	1999	Project end

UCLA, SESAME

Scalable System Software: Measurement and Evaluation

Online Repository

Measurement

Modeling

Internet

Research Community

Target Systems
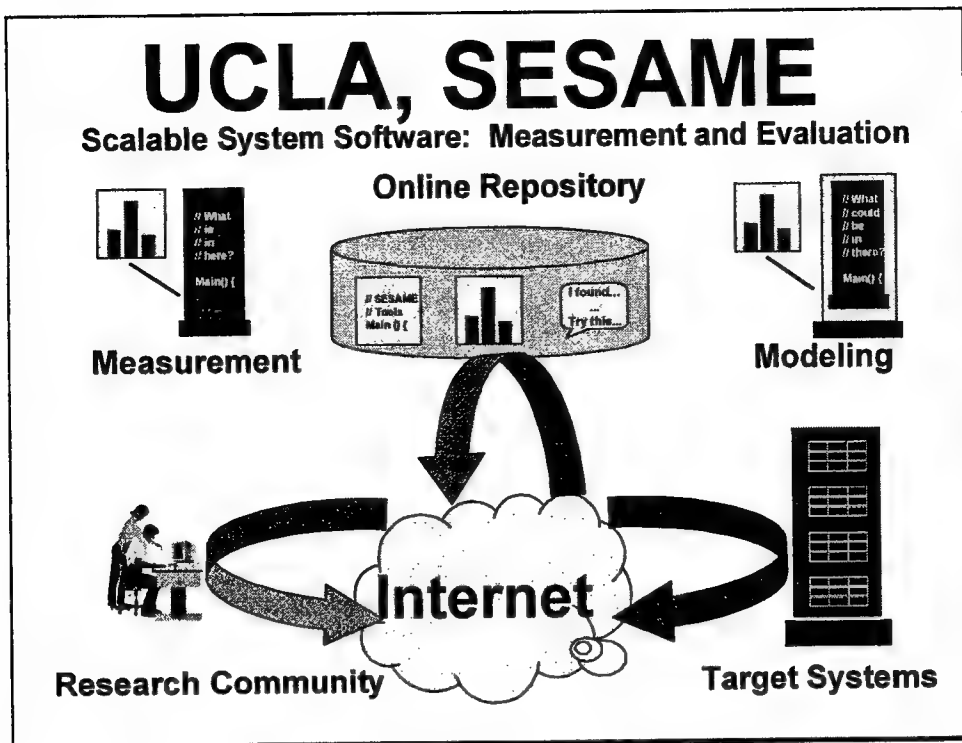
1.   INTRODUCTION

The development of modern operating systems is hampered by a disconnect within the community. Developers, users, and researchers are not sharing or coordinating the knowledge and tools they develop. Complex or novel systems using parallel or distributed architectures are especially affected. Their components are highly interdependent and must be tuned to each other. But when seeking the knowledge to do that tuning, the disconnect often leaves us with nothing more than needlessly repetitive work, rumor, folklore, or our own guesses.

This project attempts to overcome this disconnect by a coordinated approach of measurement, modeling and architectural analysis, organized through an online repository on the Web, bringing together the researchers, computers, tools, and data, and providing a place for them to interact. This project is targeted at MPP systems, but the approach and tools developed are highly flexible and applicable to other architectures.

2.   OVERVIEW

At the start of this project there was a critical shortfall in the performance of Massively Parallel Processing (MPP) systems. We were seeing performance degradations as severe as a factor of 20 in some machines; and most MPP systems delivered on the order of 5 to 10% of their peak performance across a variety of applications. It was mainly in the more mature (and therefore

more costly and cumbersome) architectures that we began to see performance losses of less than half.

Experience has shown that the MPP system architecture has much more in common with a distributed operating system than the older monolithic kernels for uniform memory architecture (UMA) machines. Unfortunately, considerable measurement, experience and corrective efforts were necessary with those distributed systems before their performance was made satisfactory; more effort than one might have reasonably expected. In our judgment, the problems principally resulted from lack of understanding of the design space, and the inapplicability of lessons learned in more conventional networking or UMA systems.

Even the design parameters of LAN-based distributed systems, however, are fundamentally different from MPP architectures. Communication delays in LAN connections are typically a minimum of several milliseconds, and effective end-to-end bandwidth is limited to tens of megabits—several orders of magnitude worse than the connections within an MPP system. As a result, the approaches used in the LAN environments appear to optimize the wrong parameters for MPP systems not only in their tuning adjustments, but also in their design and structure. Once again we are faced with design approaches, intuition and wisdom that almost certainly does not apply, or at least not directly.

Prior to this project, some glaring MPP problem areas had already been identified: for example, failure to deliver anywhere near the full bandwidth performance of hardware I/O subsystems (one commercial system ran at one-tenth the capacity of the rated hardware I/O stream); the interaction between dynamic processor allocation and more static positioning of information on disk; and crippling inefficiencies in internal operating system services. But each of these cases had initially been a surprise to systems developers. Encountering these problems serially, without prior insight, is an inordinately painful, inefficient approach that consumes far more elapsed time than necessary.

The SESAME project tests a method to address this critical shortfall in performance through a coordinated measurement, modeling, simulation, architectural evaluation, and experimental alteration effort—taking a global view of the overall systems software environment. Our goal is to probe large-scale parallel machines at the system level (i.e., operating system and compiler levels) in order to guide systems software and hardware designers.

The two main tasks are: characterizing the performance of an operating MPP, and evaluating the effect of (software and hardware) design modifications. The former requires systematic measurements and analysis of MPP environments to understand how the system is performing, why it behaves as it does and how it can be improved. The latter requires a sophisticated modeling and analysis capability to predict the effect of adjustments in design parameters that include issues of scaling and architectural modification.

To maximize success, we assembled a team that possessed skills in each of these areas, and made all results available to the larger community. In this way, we hoped to ensure coordination among the several necessary aspects of the research and to make results and tools rapidly and generally available.

3. COMPONENTS

The three major components of this approach are:

1. Tools to measure activity and performance inside the operating system of a running MPP machine.

2. A simulation environment that allows the configuration and performance characteristics of the machine to be adjusted, and the results predicted.

3. An online repository to house the tools and data, along with programs for data analysis, and a team to coordinate the research.

3.1 Measurement

Measurement of current system performance is obviously critical. A major innovation of the SESAME project has been the development of a flexible method for extracting performance measurements from inside the operating system kernel of a running system. This consists of a portable set of data transport and measurement control software, and system-specific "hooks" placed directly into the OS code. These hooks permit us to obtain performance information from much deeper in the system than would normally be possible, with a much more specific understanding of what is being measured. This system is not dependent on any special system conditions, any normal activity can be measured, and it has minimal impact of system performance.

The portable components of this instrumentation, and the method for connecting the hooks, constitute a set that is particularly useful in MPP systems, but is applicable to single systems and distributed systems as well.

Normal application-level benchmarks are used to take conventional user performance measurements and to drive the system through performance phases for measurement with the kernel tools.

3.2 Modeling

The insight derived from the detailed measurement of MPP performance will generate a number of proposals to alter the software or hardware design of future MPP systems. For this purpose we have developed a sophisticated modeling and analysis capability to evaluate the impact of these proposed changes on a large class of MPP configurations.

Most modeling efforts have a narrow focus: for instance, the simulation of a single hardware component like the processor, memory system, or interconnection network of a MPP system. Although these simulators have led to design improvements for the corresponding hardware subsystem, the impact of the modification is rarely examined in the larger context of improvements in the overall performance at the system software and at the application level. The ability to make such evaluations is a central result of the modeling and analysis effort.

Analytical models can yield useful insights into the performance of system configurations over a wide range of gross hardware characteristics and statistical workloads. However, simulations

allow us to examine the interactions among different components and to evaluate the impact of proposed changes on realistic workloads, including those derived from operational applications. Parallel execution permits simulation models that are reasonably complex to be executed for large realistic workloads with reasonable execution times.

3.3   Repository

In any complex system, the need for measurement, modeling and evaluation is critical to understanding how the system is performing, why it behaves the way it does, and how it can be improved. It is also critical to predicting how it will perform if changes are made to parameters and to the architecture, how it will scale, and how it will perform in other environments and scenarios.

This need is especially true in the case of MPP systems. However, this need is appreciated by relatively few systems designers. Even in those cases where measurement, modeling and evaluation are carried out, it is usually done "from scratch," i.e., without the benefit of tools that have been developed by other systems builders and analysts. In addition, the data that is captured and/or created is seldom kept in an "archival" form for other developers to use in other studies. The same is largely true for the tools that are developed in the project.
The only thing that remains as a reusable object are commercially supported software packages. The loss of the data, tools, and software to the community is enormous.

To address this we created a functioning MPP modeling and performance repository, populated with the data, tools, and models gathered by this project. This repository was maintained at UCLA, and included the necessary tools both for access and for adding further data to it. It can be used to encourage other researchers to use its existing contents and to add their own results.

In the repository, actual data collected from running our data collection software is combined with modeling and simulation results. We have analyzed these results and data to obtain insights into the key bottleneck areas in MPP operating system design. This integrated analysis is crucial given the radical differences between scalable architecture performance parameters and previous symmetric multiprocessing systems or distributed operating systems in LAN-based environments.

We can now make a set of proposed operating system architecture changes based on this analysis. These changes should substantially improve MPP operating systems performance, particularly in the area of parallel IO.

4

4. MEASUREMENT

4.1 Goals

4.1.1 Portability

Chief among the requirements placed upon the performance measurement software to be designed for the SESAME project is that it be readily portable across a wide variety of high-performance parallel computer systems. While no operating system-level software is ever entirely portable, the goal for SESAME was to achieve a high degree of portability so that the task of porting the measurement software would be simplified as much as practical. Equally important, this portability would ensure that subsequent performance measurements would be comparable, lending themselves to drawing valid conclusions about the impact of system architecture upon system performance. This goal affects decisions about overall performance measurement system software architecture, as well as decisions that are more closely related to the implementation of this software. An excellent technique for achieving this objective is to choose an architecture that localizes system dependencies in as few logical components as possible and that segregates truly portable software from that which must necessarily vary from one environment to another. Secondarily, it was essential that the data captured by this measurement software be in a well-established, common format to enhance the sharing of performance measurements with the high-performance computing community at large. The preferred method for achieving this goal was to select an existing data format, preferably one for which performance analysis tools already exist, increasing the transfer of technology both into and out of the SESAME project.

4.1.2 Flexibility

Many performance measurement systems are severely limited by the nature of the performance data they are capable of gathering. This limitation is primarily a consequence of the performance measurement software's inflexibility and its inability to be adapted to measurements not specifically incorporated into the design of the measurement software. To avoid this reduction in utility, the SESAME performance measurement design adopted the goal of maximizing the flexibility of that software. One way of accomplishing this was to regard the design of the measurements proper as external to the design of the measurement system. That is to say, the measurement system may be designed to consist only of a substrate of essential services upon which specific measurements may be added independent of the measurement system design and without regard to the design of unrelated measurements previously implemented. In this way, as new performance measurements are identified as being desirable, they may be incorporated into the measurement software by means of continuous augmentation without incurring redesign costs. The goal for the SESAME performance measurement system was first to design and implement this infrastructure of essential services at a relatively high level of abstraction. Once this was accomplished, the goals became to separately design and implement subsystems that completely encapsulated the details of the specific measurements, never going beneath the level of abstraction offered by the measurement infrastructure.

4.1.3 Performance

All performance measurement systems are burdened by an obligation to achieve high performance of their own software. This is an absolute necessity if the captured performance data

of the system being measured is to have any meaning. The challenge is found in the perpetual struggle between a desire for more detail in the performance data and the fact that the activity of gathering more data always perturbs the measured system to a greater degree than gathering less data. In high-performance computing systems, this is especially notable in the amount of I/O that must be done to capture the performance data and to export it from the system being measured. Some performance measurement models that work very satisfactorily at the user level, even to the degree of offering detailed traces of individual events, are entirely inappropriate at the kernel level. The goal for the SESAME performance measurement software was to design a system which would, under normal use, capture an acceptable amount of detail in its measurements while maintaining very low perturbations of the measured system—and yet under extraordinary conditions could be modified in its behavior to generate more detailed measurements at the sacrifice of system performance.

4.2   Approach

4.2.1   Prototype

As is typical of research projects that advance the state of the art, SESAME began with several questions and only a few tentative answers. Specifically, a number of issues integral to the design of the intended performance measurement software required careful definition before a satisfactory system could be created. Some of these answers were suggested by the results of prior research in parallel performance instrumentation. However, since the precise set of objectives for the SESAME project had never before been attempted, the remaining issues could be resolved only by empirical methods. To this end, the initial implementation of a measurement tool for SESAME was an experimental prototype of the eventual design selected for use with the Intel Paragon operating system.

Several implementation aspects of this prototype were deliberately chosen to be simplifications of design ideas intended for the eventual performance measurement system, for reasons of expediency. For example, the performance data output format was specific to the instrumentation code. While this was quick and easy to implement, it had the undesirable result that performance data produced by the prototype did not lend itself readily to subsequent analysis. This experience underscored the need for adopting a more flexible and portable data format. Also, for the sake of simplicity, this prototype instrumentation system was customized to the Paragon operating system and was not portable to other parallel computer architectures. The process of constructing and deploying the prototype software aided in the identification of areas in which portability needed to be emphasized in the final product. One of these lessons resulted from the fact that in the prototype, while new performance instrumentation points and data types could be added to the existing software, this required modification of *all* portions of the performance measurement system.

The functional decomposition of the resulting performance instrumentation system follows the same general lines established for the prototype and served as a viable proof-of-concept for its design. However, one very important design change arose from an unexpected defect in the prototype. Initially, both instrumentation control and data capture capabilities were integrated into a single (and at this point, nonportable) user interface. This was lacking in requisite flexibility and also exhibited a tendency to behave as a performance limitation. This experience with the prototype led directly to the resulting design, in which control and final data capture functions are specialized components.

4.2.2   Measurement Model

The primary model selected for the SESAME performance measurement system is that of statistical event summaries. In this model, events are arbitrary occurrences (from the point of view of the measurement system), but the performance data capture software accumulates a summary of information based on a sequence of events, rather than detailed data about each individual event. These summaries can take on many forms, such as lifetime averages, sliding-window averages, maxima and minima, range counts, etc. This model has numerous advantages in support of SESAME performance measurement goals. Most notably, it enhances measurement system performance by reducing the volume of data produced. The biggest disadvantage to conducting performance measurement by this model is that it is difficult to perceive patterns of performance resulting from interaction across threads of control.

The limitations of the statistical event summary model are mitigated by the optional amendment of summaries with the presence of event traces. Again, events may be any arbitrary occurrences within the instrumented software. The data capture system need not have any notion of the nature of the event, so long as it is supplied with an identification or label for the event. Upon the occurrence of each event, the performance measurement software produces an *event record*, a collection of coordinated and relevant data describing the event: what it is, where it is, when it occurred, and any other data supplied by the instrumented software through the interface points to the data capture system. Event tracing systems have a mixture of advantages and disadvantages. Event traces produce the most complete picture of the activity of even the most complex software systems. Such a model provides the only certain *portable* method for capturing time-ordered interaction in complex, multi-threaded and/or multi-processor hosted software systems. The principal disadvantage of this model is the potentially large volume of data produced. Not only is this a concern for perturbation of the system being measured, but it entails additional problems such as transport, storage, and analysis of the large amounts of performance data collected. Event tracing systems are therefore difficult to design and implement effectively. They often require the invention of custom data analysis systems to complement them.

The SESAME performance measurement software makes use of both of these models to maximize their relative advantages and minimize their disadvantages. While statistical event summaries are relied upon as the primary model, the user may select to enable event tracing in specific instances to gather more detail. To enhance both portability and flexibility of this measurement software, events are organized into related groups, called *subsystems*. Each subsystem consists of one or more related events and the data structures from which a summary record is produced. Each subsystem may be modified and manipulated independently of each other subsystem.

The format of the data generated by the SESAME performance measurement software is the popular SDDF (Self-Defining Data Format), which was pioneered at the University of Illinois at Urbana-Champaign (UIUC) under a DARPA contract. The flexibility of this format directly benefits the flexibility of the SESAME measurement system. Generating data in SDDF has the added benefit of permitting immediate use of existing analytical tools created by the UIUC *Pablo* project.

4.2.3 Data Transport

Performance measurement systems often generate large volumes of performance data, and thus the mechanism used to deliver this data to the point of analysis is an important design issue. Performance data may either be *pushed* or *pulled* off the system under examination:

- Data is *pulled* off the system under test when some part of the data reception software (that part of the data capture system not resident on the measured system) initiates the data transfer.

- Data is *pushed* off the system under test when data transfer is initiated by the data capture software that resides on the measured system. This typically occurs when required to prevent the overflow of performance data buffers.

As a general rule, the statistical event summary model of performance measurement generates fixed amounts of data; the data changes over time but the volume of data does not. In this case, pulling data off the system under test is a good solution since it minimizes the perturbation of the system. In contrast, the event-tracing model generates a continuously growing body of data as long as performance measurement is active. SESAME is designed to measure the performance of a continuously running system, i.e., an operating system, and it also supports optional event tracing. Thus, a hybrid design has been chosen for SESAME. In the ordinary case, when only event summaries are enabled, the captured performance data is pulled off the measured system. Only when event tracing is enabled does the data capture software push the gathered data off the system, and then only when an event trace buffer has filled. Event trace data may also be pulled off the measured system, such as in the situation where event tracing is purposely limited in scope and the trace buffer is chosen to be large enough that it is not necessary to push event trace data off the system while measurements are being conducted. This mechanism is most advantageous in terms of portability and flexibility, and likewise minimizes perturbations induced on the measured system by the performance measurement software.

4.2.4 Control

The SESAME performance measurement system contains three interfaces by which the instrumentation may be controlled:

- A world-wide web (WWW)-based graphical user interface.
- A simple command-line interface.
- An application programming interface which may be incorporated into any application executing on the instrumented system.

For each of these control interfaces, a rich hierarchy of control mechanisms is provided:

- The entire instrumentation system may be enabled or disabled, as with a simple on/off switch.

- Each instrumentation subsystem may be individually and selectively enabled or disabled, affecting only those instrumentation points within the subsystem.

- Each separate instrumentation point may be individually and selectively set to be inactive, to contribute to the subsystem's summary data, or to produce event traces in addition to contributing to event summary data.

Any or all of these control modes may be exercised over a single processing element (node), all nodes, or any subset of them. Likewise, it is a simple matter to exercise control over an individual instrumentation subsystem, all subsystems, or user-designated subsets of subsystems. These interfaces also provide direct control over performance instrumentation system global parameters, such as the destination of resultant performance measurements and the rate at which data is generated.

4.2.5 Decomposition

The SESAME measurement software consists of five major components:
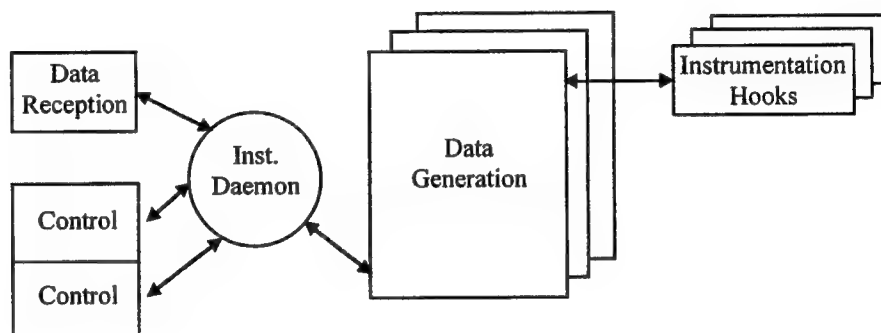
The *instrumentation hooks* and *data generation* components both reside on the system being measured. Together they are responsible for the creation and capture of performance data during execution of the measured system. In a multicomputer system, this software has several copies, each executing on a different processing element (node). The *instrumentation daemon* acts to link the *control* and *data reception* components with *data generation*. It executes as a user program on a single processing element of the measured system. At appropriate times during execution, the performance data is moved to the *data reception* component. This software, residing on a platform external to the system being measured, passively receives data and stores it directly into an output data file for subsequent analysis. The final component is the *control* mechanism. Its sole purpose is to execute control directives received from the external control inputs and to respond to inquiries about the current state of the instrumentation system. The *control* mechanism is distributed across several platforms.

Each instrumentation hook consists of nothing more than a few lines of code which are inserted directly into the software being measured. These insertions are the only changes to the measured software itself. As experience with a given system being measured grows, instrumentation hooks are added by the researcher to examine specific performance areas. Execution of a hook creates an entry point into the data generation software component, passing as arguments relevant items of data, as defined for the instrumentation point. For reasons of portability, the semantics and function of these interface routines remain invariant when the instrumentation system software is ported from one host environment to another. Further, only the arguments passed into the data generation component from the execution of a hook identify the nature of the region of measured code; the same interface routines are used uniformly throughout the measured software. This design creates an identity between the passage of execution through the instrumentation point

with the occurrence of an event. This also permits adequate flexibility so that events remain arbitrary (viewed from the perspective of the measurement system) and instrumentation may be expanded into other areas of the measured software without modifying the measurement system.

At the core of the data generation component is an instrumentation infrastructure, which is divided into portable and non-portable sections. It includes items such as obtaining system clock values for time-stamps, storage allocation and deallocation primitives, semaphores or other priority locking mechanisms to protect critical sections of code, performance data buffer management, and low-level interface routines managing communication with other components of the instrumentation system. It provides the minimal expected subset of system resources needed to support the instrumentation data generation component. These routines promote portability by isolating the rest of the measurement software from system-specific code. This further enhances flexibility of the instrumentation system in that new instrumentation points and subsystems may be created or modified without alteration of this core subcomponent.

The data reception component resides on a platform that is external to the system being measured. Its primary function is reception and temporary storage of performance data transferred from the instrumentation daemon. Secondarily, it is used to forward performance data to the repository for long-term storage and analysis. Because of its simple functionality, the data reception component is highly portable, independent of the type of system being instrumented, and is not bound by any location requirements to the measured system.

The control component provides a path for both user and application selection of the various operational modes of the instrumentation system:
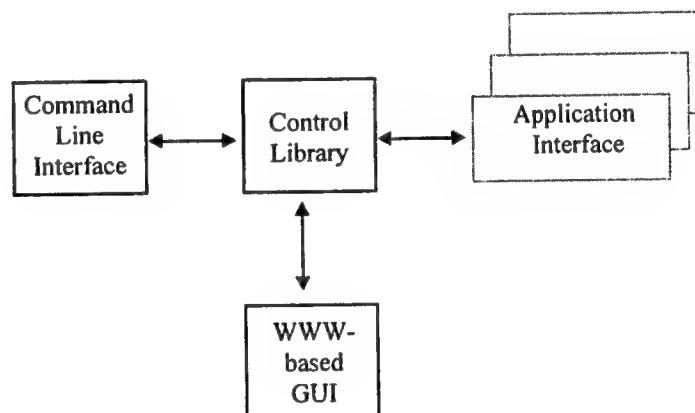
- , he *control library* constitutes the core of the control mechanism. All commands are received there and relayed to the instrumentation daemon for processing. Similarly, instrumentation status information always passes back out through these routines. The control library includes the application interface and is used by all three interface mechanisms.

- The *command line interface* is a simple command and status mechanism, which executes on the system being measured. It utilizes the application interface of the control library.

- The *WWW-based GUI* provides the primary control mechanism and satisfies the need for remote instrumentation control.

- Applications executing on the system being measured may optionally be linked with the *application interface* of the control library and thus have direct control over the instrumentation system.

4.2.6   Implemented Measurements

Through experimentation and collaboration with both the repository and modeling groups in the SESAME project, the measurement group designed and implemented a rich and diverse set of instrumentation subsystems for measuring Intel Paragon operating system performance. As the measurement system reached maturity, identification and creation of these subsystems decreased in difficulty, even as the subsystems themselves grew more sophisticated. The subsystems described below are conceptually portable to other parallel computer architectures, although their relative significance may increase or decrease, depending on implementation details of the operating system being instrumented.

Processor Utilization

The processor utilization instrumentation subsystem was created to gather and report measurements of the relative fractions of system, user, and idle time. This subsystem is flexible enough to report this data for more than one processor per node, as the Intel Paragon is capable of using two processors (not counting a message coprocessor) on each node. This subsystem is unique in that it requires *no* instrumentation hooks. In other words, this instrumentation subsystem entailed absolutely no modification of existing operating system code. A direct consequence of the absence of hooks is the absence of trace records; this subsystem produces only the statistical summary type of performance data record, indicating processor utilization over a time interval. This implementation introduces an exceptionally low (virtually unmeasurable) perturbation on the operating system, and thus gives an accurate first-order indication of system performance.

Task Management

The task management instrumentation subsystem was designed to measure task creation and task termination events. In addition to the capability of creating trace records for each of these two events, this subsystem gathers counts of these events over time intervals and periodically reports this data in a summary record. This metric can be a critical measurement of operating system performance in cases where the system is heavily loaded by the activity of task creation or termination.

Virtual Memory Management

The virtual memory (VM) management instrumentation subsystem was the first truly sophisticated performance measurement suite created for SESAME. It provides a wealth of coordinated data on six specific page-faulting activities within the virtual memory management code of the operating system. This is especially important for the Paragon operating system, since all file system activity takes place through VM mappings. In addition to categorizing and optionally issuing trace records for six specific kinds of VM page faults (page-ins, page-outs, page-reactivation faults, zero-fill page faults, copy-on-write faults, and total page faults) each of these events is classified as corresponding to ordinary file I/O activity, executable file activity, or non-file VM activity that is internal to the Paragon microkernel. This subsystem tallies both event counts and event duration time intervals for each of these event subdivisions and reports all of this data correlated in a single subsystem statistical summary record.

11

Interprocessor Communication

The interprocessor communication (IPC) instrumentation subsystem measures overall IPC message-passing activity within the operating system. Three basic events are instrumented: message sends and receives as implemented in the operating system software, and the completion of the interprocessor message send at the hardware interface level of detail. Trace records for each event may be issued by the subsystem, if desired, although this is strongly discouraged on the Intel Paragon since the instrumented code executes on the message coprocessor. Each event is further categorized by message size group, the number and limits of the groups being configurable parameters of the IPC instrumentation subsystem. This subsystem tallies both event counts and event duration time intervals for each of these event subdivisions and reports all of this data correlated in a single subsystem statistical summary record.

Parallel File System

The parallel file system (PFS) instrumentation subsystem measures PFS I/O activity within the operating system. Conceptually, any number of PFS events could be instrumented, but on the Intel Paragon this is limited to PFS read and write activity as a consequence of operating system software architecture. This subsystem further categorizes PFS read and write events by the size of the I/O operation. The number and limits of size groups are configurable parameters of the PFS instrumentation subsystem. In addition to the capability of generating trace records for these two basic events, this subsystem tallies both event counts and event duration time intervals for each of these event subdivisions and reports all of this data correlated in a single subsystem statistical summary record.

4.3 Results

4.3.1 Portability

Any truly portable system performance measurement system requires access to and modification of the operating system at the source code level. In the case of commercial operating systems, it is absolutely necessary to have the cooperation, if not the active participation, of the corporation controlling the rights to their proprietary source code. The SESAME project was fortunate to obtain this degree of access to the Intel Paragon, a widely used high-performance parallel machine at the time this project was first envisioned. Among the very highest of requirements placed upon the SESAME performance instrumentation software design was the specific goal of portability across parallel system architectures. Internal code reviews indicate that this goal has been achieved, subject only to necessary limits imposed by design functionality. However, the truest test of software portability is to actually port the software to another machine. One of the very few project goals not achieved was to perform this port. Contrary to expectations, the SESAME project was unable to obtain access to another major parallel operating system's source code. Despite this shortcoming, the SESAME project members believe they have accomplished the goal of designing and implementing a portable performance instrumentation system for parallel operating systems.

4.3.2 Flexibility

The SESAME performance measurement system excels in its realization of flexible, extensible, easily modifiable instrumentation. By virtue of its modular design, all of the implemented performance measurement subsystems were created subsequent to completing the implementation

of the core components of the measurement software, those components that are independent of specific measurements. Experience with early instrumentation subsystems suggested enhancements that were made to the measurement system, simplifying the addition of further subsystems. The final two subsystems were implemented with no changes whatsoever to the remainder of the performance measurement system, even though these subsystems are the most sophisticated ones designed for the SESAME project. As completed, the rate-determining factor in the creation of new measurements is the complex process of *identification* of what specific measurements are to be made and where in the existing operating system source code they can be instrumented. The flexibility of the SESAME performance measurement software permits design and implementation of new measurements to be accomplished very rapidly and easily following identification of those measurements.

4.3.3 Performance

Given the large number of controls that the SESAME instrumentation software makes available over its own behavior and the broad range of system performance regimes in which the instrumentation software is designed to operate, it is very difficult to call out specific numbers as accurate measurements of overall instrumentation system performance. The SESAME project measurement team has performed some baseline comparisons of system behavior without instrumentation present and with it present in various configurations.

When no event tracing is being done, the SESAME instrumentation software performs very well. The dominant factors are primarily the rate at which measurement subsystems are generating statistical summary data records (controlled by a factor called the *pull interval*), and secondarily the number of measurement subsystems and therefore instrumentation hooks which are active. At typical pull intervals—ten seconds or more—the Intel Paragon operating system displays aggregate instrumentation perturbation ratios less than 0.1 percent, with the measured value dropping in inverse proportion to the pull interval. In certain cases (such as only the processor utilization measurement subsystem being active and pull intervals in excess of 30 seconds) the induced instrumentation perturbations are not measurable by the system clock precision and are effectively absent.

This is not to say that the SESAME instrumentation software cannot be made to exhibit large perturbations of the operating system software. Activating event traces in some of the subsystems on a large number of nodes can quickly lead to the production of performance data at such a high rate that the operating system is overwhelmed by the demands of the performance measurement activity. In particular, on the Intel Paragon it is a message coprocessor that performs the interprocessor communication, and the critical path algorithms executing on that processor are highly sensitive to perturbations induced by measuring its performance. As the execution of event hooks is localized to the same code which is being measured, activating event traces for the IPC measurement subsystem can rapidly lead to catastrophic system failure, as has been observed in testing the SESAME instrumentation software. The only real solution is to use event tracing only with great discretion, if at all.

These observations of bimodal behavior (outstanding performance in one basic operational mode, abysmal performance in another) indicate that the design and implementation of the SESAME performance measurement system was both sound and well executed. The concept of relying primarily upon constant data rate statistical event summaries in independently controllable performance measurement subsystems is validated by experience in practical application.

5. MODELING

5.1 Overview of the Simulation Tool

As part of the Sesame project, a simulator designed to model the performance of MPI and MPI-IO programs was built. Simulators for parallel programs can be used to test, debug and predict the performance of parallel programs for a variety of parallel architectures. Most existing simulators (Brewer et al. 1991, Davis et al. 1991, Covington et al. 1991) use direct execution to simulate the sequential blocks of code, and simulate only the communication and/or I/O events. As sequential execution of such models (Legedza and Weihl 1996, Reinhardt et al. 1993, Dickens et al. 1994, Dickens et al 1996) are typically slow (slowdown factors of 2 to 15 per processor are not atypical), several researchers have used parallel execution of such models with varying degrees of success. The primary difficulty in obtaining better performance is the significant synchronization overhead in the parallel simulator. With the use of the simulator we explored novel conservative synchronization algorithms for parallel simulation of message passing parallel programs. We combined the existing null message (Misra 1986) and conditional event (Chandy and Sherman 1989) protocols together with a number of optimizations to significantly reduce the frequency and cost of synchronizations in the parallel simulator. An existing MPI (MPI Forum 1993) program may be linked with the MPI-SIM library (after an appropriate pre-processing stage described subsequently) to predict its performance as a function of the desired architectural characteristics; a programmer is not required to make any modifications to the original MPI program. We have also incorporated a simulator to predict the performance of MPI-IO programs (Bagrodia et al. 1997). The simulator can take into account various architectural characteristics (e.g., number of I/O nodes, message passing latency), file system characteristics (e.g., data caching and partitioning), and secondary storage device characteristics (e.g., seek time, data transfer time). Using the simulator we predicted the performance of three benchmark applications including the BTIO benchmark from the NAS benchmark suite (Bailey et al. 1995). We studied the performance of the benchmark applications as a function of alternative implementations for collective I/O operations, including two-phase I/O (del Rosario 1993), and the performance of the benchmark applications as a function of alternative caching strategies including cooperative caching (Dahlin et al. 1994).

5.2 MPI Simulation Model

5.2.1 MPI Overview and Core Functions

MPI (MPI Forum 1993) is a message-passing library which offers a host of point-to-point and collective interprocess communication functions to a set of single-threaded processes executing in parallel. All communication is performed using a **communicator**—which describes the group of communicating processes. Only member processes may use a given communicator. In the subset of MPI we simulate, all collective communication functions are implemented in terms of point-to-point communication functions, and all point-to-point communication functions are implemented using a set of *core* non-blocking MPI functions. The core functions include MPI_Issend, a *non-blocking synchronous send*, MPI_Ibsend, a *non-blocking buffered send*, MPI_Irecv *non-blocking receive* and MPI_Wait.

14

The primary difference between the two sends is that the synchronous send completes only when the receiver has accepted the message using a matching receive; the buffered send completes as soon as the data has been copied to a local buffer. The buffer space is released only when the data has been transmitted to the receiver via a synchronous send. Each point-to-point MPI message carries a *tag* and the *sender-id*. A receive may be selective, accepting a message only from a given sender and/or with a given tag. Alternately, it may use wild card arguments, MPI_ANY_SOURCE or MPI_ANY_TAG, to indicate that a message from any source process or with any tag value is acceptable. The wait is simply a function which blocks the process until the specified non-blocking (send or receive) operation has completed.

We use the terms *target program* to refer to the MPI program whose performance is to be predicted, *target machine* as the machine on which the target program executes, *simulator* as the program that simulates execution of the target program on the target machine, and *host machine* as the machine on which the simulator executes. In general, the host machine may be sequential or parallel. For direct execution, it is important that the processor configurations in the host and target machine be similar.

5.2.2   Preprocessing MPI Programs for MPI-SIM

In general, the host machine will have fewer processors than the target machine (for sequential simulation, the host machine has only one processor); this requires that the simulator provides the capability for multi-threaded execution. Since MPI programs execute as a collection of single-threaded processes, it is necessary to provide a capability for multi-threaded execution of MPI programs in MPI-SIM. We have developed MPI-LITE, a portable library to support multi-threaded MPI programs.

Executing an existing MPI program as a multi-threaded program requires additional modifications. The primary one deals with transforming the permanent variable, i.e., global variables and static variables within functions. If the unmodified MPI program is executed as a multithreaded program, all threads on a given host process will access a single copy of each permanent variable. To prevent this, it is necessary to *privatize* the permanent variable such that each thread has a local copy. Each permanent variable is redeclared with an additional dimension whose size is equal to the maximum number of threads in a host process. Each reference to the permanent variable is also modified, such that each thread uses its id to access its own copy of the permanent variable. This process of adding a dimension to the permanent variables is referred to as *privatization*. A preprocessor is provided with MPI-SIM that automatically privatizes permanent variables, converts each MPI call to the corresponding MPI-SIM call, and implements miscellaneous transformations needed to link the program with the MPI-SIM library. In MPI-SIM the routines for inter-thread communication are syntactically identical to those for inter-process communication, except for the use of a special prefix to distinguish between the two.

5.2.3   Simulation Model for Core Functions

We present a model for execution and simulation of the four core functions. The simulation model defines a logical process (LP) for each process in the target program. Each LP has a message queue for each communicator of which the LP is a member, a simulation clock, and an ordered list (ordered by simulation timestamp) of the pending (send and receive) operations of the LP; this list is referred to as the *request list*. Simulation of a process in the target program by a corresponding LP in the simulator proceeds as follows: sequential code blocks are simulated via direct execution; each call to an MPI communication statement (collective or point-to-point) is translated to a call to the corresponding MPI-SIM function; MPI-SIM internally implements each

15

call to a collective function in terms of the core communication commands described in Section 5.2.1. For brevity, we will not describe the translation in this report; the reader is referred to (Prakash 1996). We will briefly describe the simulation of the core commands.

The sends in the MPI core are simulated by sending a message (with source, destination, tag, communicator and data) to the receiver LP. The message is timestamped with the send timestamp (the current simulation time of the sending LP) and the receive timestamp (which is the send timestamp plus the predicted message latency). For buffered sends, the overheads and functionality for buffer availability check are included in the simulation. The simulation of MPI_Irecv simply adds a request to the request list. The action taken for the wait depends on the type of the specified operation. For instance, for wait on a receive operation, the LP is blocked until a matched message is available. Of course, the LP must remove messages in the order of their simulation timestamps and *not* in the order in which messages are physically deposited in its queue. When an appropriate matching message is removed, the LP's simulation clock is updated to the maximum of the current simulation time and the receive timestamp of the matching message, an acknowledgment is sent to the sender, and the LP is resumed. For the synchronous send operation, the LP blocks until the corresponding acknowledgment has been received from the destination. At this time, the simulation time of the LP is updated to the maximum of the current simulation time and the receive timestamp of the acknowledgment.

5.3 Parallel I/O and Parallel File Systems

Parallel file systems offer the high performance necessary for running complete parallel scientific applications. To provide maximum scalabilty, the machine's processors are generally divided into two separate groups: compute nodes (cnodes) and I/O nodes (ionodes). Cnodes run the user applications and send I/O requests to the ionodes. All disk access is performed by the ionodes, which each have their own secondary storage device (SSD) and manage a subset of the file system's total data. In this manner, the parallelism of the user application can be preserved when performing I/O. Some parallel file systems (Corbett and Feitelson 1995, Corbett and Feitelson 1996) even allow user processes to configure the underlying I/O parallelism to match their intended access pattern, further enhancing performance.

When properly tuned, the use of file system caching can drastically reduce the number of I/O requests that must wait to be serviced by the SSD. Unfortunately, choosing the correct cache configuration for peak performance is no mean feat. This is particularly true of parallel file systems which offer a wide variety of techniques for managing the many caches spread across the ionodes and cnodes. One technique with much potential is *cooperative caching*. Presented in (Dahlin et al. 1994) as a set of high-performance caching algorithms for use within a network file system, cooperative caching attempts to improve performance through better management of multiple client and server caches. This basically includes the capability to retrieve data from remote client caches and the global control of a portion of the client caches. With the low latency provided by the interconnection network of a parallel machine, cooperative caching may be even better suited for use within a parallel file system.

For users requiring maximum I/O performance, the pairing of a high performance parallel file system with a language-level parallel I/O library, such as MPI-IO, is a natural choice. MPI is a standard message-passing library that provides a number of point-to-point and collective communication primitives (MPI Forum). MPI-IO is a proposed extension to MPI that would incorporate parallel I/O constructs (MPI-IO committee). The proposed constructs include both

independent and collective I/O operations, asynchronous I/O calls, file access via independent file pointers, shared file pointers and explicit offsets, and local and distributed datatype constructors.

The goal of our simulator is to provide a flexible environment for studying the performance, interaction, and design tradeoffs inherent with the use of high performance parallel file systems and parallel I/O libraries. In addition, support for parallel execution of the simulator will considerably reduce the turnaround time for the execution of detailed simulation models for complex system configurations and workloads.

5.4   Parallel I/O Simulator

The simulator contains the following primary components:

4   **Simulation kernel**: Execution of a discrete-event model requires a kernel that can execute the events in their timestamp order. The simulation kernel provides sequential simulation for use on a uniprocessor, as well as parallel simulation using multiple synchronization protocols for a parallel architecture.

- **MPISIM**: The simulation of MPI-IO commands requires the ability to simulate MPI communication commands. MPISIM is a multi-threaded MPI simulator that was developed for this purpose (Prakash 1996, Prakash 1998).

- **PIO-SIM**: simulates the individual and collective I/O constructs provided by MPI-IO. These constructs include creating, opening, closing and deleting a file; most data access (read/write) operations; and a local datatype constructor introduced as part of the MPI-IO specification. The collective data access calls have been implemented in the simulator using a number of alternative implementations, which are described subsequently.

- **PFS-SIM**: simulates the parallel file system used to service I/O requests generated by the MPI-IO programs. This component is self-contained and may be replaced by a simple disk access model in order to speed up simulation whenever a detailed system model is not required. However, use of PFS-SIM allows the study of a wide variety of parallel file system configurations.

- **Secondary Storage Device models**: The behavior of the physical disks simulated by a set of disk models. We have included simple models based on seek time, rotational latency, and data transfer rate, as well as a highly detailed model developed at Dartmouth (Nieuwejaar and Kotz 1996).

The I/O simulator has been designed to be both modular and extensible. It is relatively easy to replace individual modules at each of the preceding levels. In particular, it is straightforward to replace the SSD models, to modify the caching or partitioning policies used by PFS-SIM, to modify the implementation of a specific collective MPI-IO operation supported by PIO-SIM, and to change the model of the interconnection network.

We assume that the target program includes local code blocks that are simulated by direct execution, MPI communication calls that are simulated by MPISIM, and MPI-IO commands that are handled by PIO-SIM. Each process of the target program is modeled by a single thread in the simulator; we refer to this thread as the *target LP*. When a target LP executes an MPI-IO call, it is intercepted by PIO-SIM. In the case of collective I/O, MPISIM's underlying communication

17

facility is used for synchronization and communication between target LPs. If complex user-defined datatypes are used (which allow processes to access non-contiguous pieces of data with a single MPI-IO call), the single non-contiguous request is decomposed into multiple contiguous requests. PIO-SIM uses standard UNIX system I/O calls (e.g., read(), write(), etc.) to replicate the functionality of these operations in the simulator. These I/O requests are then passed to PFS-SIM, in order to determine I/O execution time for the simulated subsystem.

The I/O subsystem can be simulated at multiple levels of detail. At the most abstract level, a simple analytical model is provided which calculates the I/O time as a function of specified disk performance characteristics and the size of the data transfer. For more detailed analysis, PFS-SIM is used to simulate the parallel file system's cnodes, ionodes and disks, using separate LPs to represent each of these entities. The I/O requests from PIO-SIM are passed to the cnode LP corresponding to the target machine's cnode on which the requesting target process would be running. Each request is then distributed to one or more ionode LPs based upon the physical data layout selected. Similarly, ionode LPs send their requests to disk LPs, where I/O service times are calculated. This path is then reversed, completing the I/O simulation. When caching is included in the simulation model, this path may of course be shortened or may include additional communication among ionode LPs and cnode LPs (e.g., block invalidation requests or cooperative caching messages).

5.4.1 PIO-SIM

PIO-SIM is the layer of the simulator which simulates MPI-IO calls. PIO-SIM implements all data access operations except calls using a shared file pointer. This includes positioning using explicit offsets and individual file pointers, independent and collective coordination, and blocking and nonblocking synchronism. Two central problems in the simulation of MPI-IO programs include support for user-defined datatypes and specific implementations of collective I/O operations. We address them in the following sections.

MPI-IO Datatypes

In MPI-IO, MPI datatypes are used to specify a "view" of the file. Effectively, this allows processes to execute I/O operations with different file access patterns. For example, if four processes were to access a common file, which contained a matrix stored in row-major order, each process could open a filetype that represented only a submatrix of the bigger matrix (e.g., a specific row or column). Data that lies outside of this filetype would not be accessible to this process. Subsequent read and write operations to the file would pertain only to the section of the matrix that lies within the filetype used to open the file.

Since the simulator was designed to be portable and not rely on any specific implementation of MPI (e.g., internal data structures of the MPI runtime system), the attributes of each datatype are extracted by PIO-SIM when a call is made to a datatype constructor. This information is then stored within the simulator so that subsequent calls that use this datatype can refer to it as needed. Consequently, PIO-SIM maintains its own internal structure to represent datatypes. This structure consists of a list of base offsets and the length (number of bytes) of valid data at that offset. This structure is similar to I/O lists—as found in PMPIO, NAS's implementation of MPI-IO (Fineberg et al. 1996), as well as type maps—as found in the MPI-IO implementation for the NEC Cenju-3 (Sanders et al. 1996). We will refer to this structure simply as an IOL.

In MPI-IO, opening a file is a collective operation. Upon invocation, each target LP creates an IOL to represent its filetype (ftype IOL). Afterwards, each target LP exchanges its ftype IOL with all other target LPs. This is done to facilitate using global knowledge about any subsequent collective read/write operation for optimized collective I/O.

Finally, during a read or write call, the target process specifies a data buffer as well as its corresponding datatype (btype). A btype IOL is constructed at each call so that the data from the user buffer can be mapped to the correct locations within the file according to the ftype IOL. The mapping requires each target LP to create a request IOL that represents the entire I/O request. This results in using multiple (as well as partial) copies of the ftype IOL to "tile" the entire request. Subsequently, the I/O operation is performed by the target LP by traversing the request IOL, seeking out the base offset, and reading/writing the number of bytes as specified in each element of the request IOL.

Collective I/O Implementations

In the case of collective I/O operations, the underlying implementation of the collective I/O algorithm is hidden from the user. The basic idea behind collective I/O algorithms is to use global knowledge about the I/O transfer among the processes in order to combine many small requests into fewer large requests, thereby reducing disk access overhead.

PIO-SIM currently supports the following collective I/O implementations:

- **Global Barrier Collective I/O.** Upon making the collective I/O call, all processes execute a barrier to synchronize at the outset, and then proceed with the regular, non-collective call. This form of collective I/O does not optimize the requests; it only serves as the base implementation that is allowed by the MPI-IO specification. This is the approach taken by versions 1.0 through 1.2 of MPI-IO/PIOFS, the implementation of MPI-IO by IBM for the SP2.2

- **Node Grouping.** Processes participating in the collective call are partitioned into groups and the I/O requests for each group are implemented in turn. Thus node grouping attempts to avoid flooding the system with too many simultaneous requests. Its utility is heavily dependent on the application (Nitzberg 1992).

- **Two-Phase I/O.** I/O is done in two phases (Rosario et al. 1993). In the first phase, processes arrange their requests according to a conforming distribution. This is a permutation of the data across the processes so that it coincides as much as possible with the underlying file layout. The actual I/O is done in the second phase to reduce overall disk traffic. The I/O requests are optimized in two ways: 1) duplicate requests are eliminated, and 2) numerous small requests which lie contiguously are combined into one large request.

5.4.2 PFS-SIM

The basic structure and functionality of PFS-SIM is taken from the Vesta parallel file system, a highly scalable, experimental file system developed by IBM (Corbett et al. 1996). Many of Vesta's features have been included in the design of PFS-SIM, most notably, the use of an interface which allows user applications to configure the parallelism actually used to perform I/O. In addition to the flexibility contained within the Vesta interface, PFS-SIM allows many of the file system's physical characteristics to be varied, such as cnode/ionode ratio, number of disk

drives attached to each ionode, and disk drive characteristics, as well as a multitude of different cache configurations.

Caching

While the Vesta file system implemented caching only at the ionodes, PFS-SIM supports systems which have caching at both ionodes and cnodes. This caching setup offers a larger variety of configurations for study, including cooperative caching. PFS-SIM also supports a full range of cache sizes, cache block sizes, cache associativities (direct-mapped, fully associative, set associative) and write policies (write-through/write-back, write-allocate/write-around). Write-invalidation is used to maintain cache coherency, though write-update could easily be added. Block replacement uses the LRU algorithm.

The cache management policies implemented by PFS-SIM are:

- **Base caching.** Provides local cache at both ionodes and cnodes, but does not utilize any of the cooperative caching techniques. Reads involve checking the local cnode cache and then the cache of the ionode(s) responsible for managing the required data. If the data is not in either of these caches, it is retrieved from disk. Writes are somewhat more complex and depend on the exact write policies chosen, but operate much the same way. Write-invalidation causes the invalidation of any remote cnode cache blocks which contain data just written.

- **Greedy forwarding.** Adds the retrieval of data from remote cn caches. Whenever an ionode receives a request for data which it is not currently caching, it checks to see if any cnode in the system is caching the data. If so, the request is forwarded to that cnode, otherwise the data is read from the disk.

- **Centrally coordinated caching.** Typically used in addition to forwarding, centrally coordinated caching attempts to improve the global cache hit rate of the system by coordinating the contents of the cnode caches. A specified portion of each cnode cache is collectively managed by the ionodes, with the remaining portion still managed locally by the cnode. Whenever a cache block is evicted from an ionode cache, it is sent to that ionode's portion of the centrally coordinated cache. This is very similar to physically moving some of the cnode cache to each of the ionodes. The penalty for this is a reduced hit rate at each cnode's local cache.

- **Optimized globally managed caching.** For the network environment in (Dhalin et al. 1994), the tradeoff of a reduced local hit rate for a higher global hit rate resulted in peak cache performance occurring when 80% of each cnode's cache was coordinated. The fact that retrieving data from a remote cnode in a parallel file system is much less expensive than retrieving data from a remote client in a network file system led us to believe that performance would continue to improve until 100% of all cache was coordinated. This would remove any data redundancy within the caches and eliminate the need for expensive cache coherency protocols. The order in which blocks are placed within the different caches was changed in an attempt to minimize the effect of coordination on local cache hit rates. Blocks read from the disk are first placed in the cache of the cnode that issued the read. The ionode cache is then used to house evicted cnode cache blocks. With 100% centrally coordinated caching, blocks would be placed first in the ionode cache and then moved to the cnode caches as they are evicted.

20

5.5   Parallel Execution of A Simulation Model

Two types of protocols have commonly been used in the parallel simulation of parallel programs: the synchronous or quantum protocol (e.g., SimOS in Rosenblum et al. 1995, Rosenblum et al. 1997), and the asynchronous protocols (e.g., LAPSE in Dickens et al. 1994). In the synchronous protocol, each LP periodically simulates its corresponding process for a previously determined interval Q, termed the *simulation quantum*, and then executes a global barrier. These barriers are used to ensure that messages from remote LPs will be accepted in their correct timestamp order. An LP waiting at a receive will accept a matching message from its buffer only if *the receive timestamp of the message is less than the simulation time at which the current quantum terminates*. If more than one such message is present, the LP will select the one with the earliest timestamp; if no such messages are present, the LP remains blocked, and its simulation time is updated to the end of the current quantum. The synchronous protocol is guaranteed to be accurate only if Q<L, where L is the communication latency of the target architecture. However, a small Q implies frequent global synchronizations, leading to poor performance. (If the host machine provides an efficient hardware implementation of global synchronization (e.g., CM5), it might be feasible to obtain good performance even with a small value of Q). Simulation efficiency can be improved by using a larger quantum; however with Q>L, it is no longer possible to guarantee that the simulator is accurate. Thus parallel simulators (e.g., SimOS) that use this protocol offer two simulation modes: fast and inaccurate, or slow and accurate.

MPI-SIM uses an asynchronous protocol, which reproduces the communication ordering of the target program in the simulator. LPs have two attributes associated with them at all times: *execution status* (blocked, running or terminated) and *simulation status* (deterministic or non-deterministic mode). An LP is *blocked* if it has executed a receive statement and no matching message is available; otherwise it is said to be *running*. An LP is in deterministic mode *if every receive request in its request list explicitly specifies the source (i.e., no receive contains MPI_ANY_SOURCE as the source)*. Each LP executes without synchronizing with other LPs until it gets blocked on some wait operation; a synchronization protocol is used to decide if the LP can or cannot proceed with a message from its buffer. We briefly describe our protocol.

Each LP in the model computes a local quantity called *earliest input time* or **EIT** (Jha and Bagrodia 1993). The EIT represents a lower bound on the receive timestamp of future messages that the LP may receive. Consequently, upon executing a wait statement, an LP can safely select a matching message with a receive timestamp less than its EIT. Different asynchronous protocols differ only in their method for computing EIT. Our implementation supports various protocols including the *Null Message Protocol (NMP)* (Chandy and Misra 1979), the *Conditional Event Protocol (CEP)* (Chandy and Sherman 1989), and a new protocol which is a combination the two (Jha and Bagrodia 1993). Due to space limitations, we have omitted details of the protocol; the interested reader is referred to Prakash (1996).

The primary overhead in implementing parallel conservative protocols is due to the communications to compute EIT and the blocking suffered by an LP that has not been able to advance its EIT. We have suggested and implemented a number of optimizations to significantly reduce the frequency and strength of synchronization in the parallel simulator, thus reducing unnecessary blocking in its execution. The primary optimizations include:

- **Automatic detection of deterministic fragments in the parallel program.** In general, an LP is blocked either if its buffer does not contain a matching message or if the timestamp on the message is greater than the LP's EIT. However, an LP in the *deterministic* mode can

21

proceed as soon as it finds a matching message, regardless of its EIT. This is an optimization within the framework of the null message protocol.

- **Reducing blocking time of an LP by exploiting the communication characteristics of the application.** By precisely defining potential message sources, an LP can reduce the communications that are used to advance its EIT.

- **Reducing the frequency of synchronization with dynamic extraction of lookahead.** Lookahead is the ability of an LP to predict lower bounds on future times at which it will *generate* a message for other LPs. Extracting tight estimates for each communicating partner leads to fewer synchronizations than the commonly used static methods for computing lookahead.

SESAME Repository
http://origin.cs.ucla.edu/
Internet Accessible Libraries For...
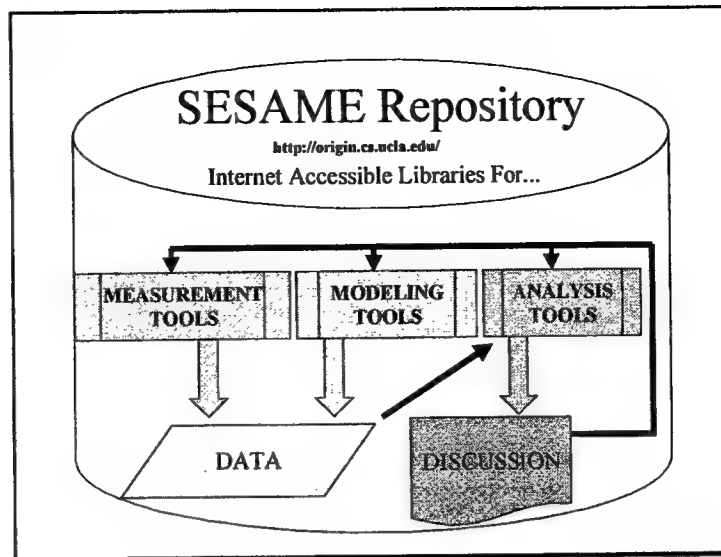
6. REPOSITORY

6.1 Overview

6.1.1 Purpose

The Repository (<http://origin.cs.ucla.edu/>) uses World Wide Web technology to serve three general roles in the SESAME project:

- House
- Share
- Coordinate

The most basic function is to house the products of the research. This includes the programs for measurement and simulation instruments, the data produced by them, the tools used to analyze the data, and the results of the analysis. Along with housing is the need to make this available to the project members and the research community at large. This involved the production of a WWW interface to all components housed. As the central point for the project, the repository also serves the coordinating role: providing a contact point, maintaining a discussion board, and serving requests. Repository staff also took primary responsibility for development of analysis software, as well as design and execution of experiments.

The WWW interface is the major innovation of the repository, and is designed into all components. While the material housed can be downloaded and used locally in a conventional manner, it is intended mainly for online use. The design goal was that the only tools a user would need to access the repository and participate in the research would be an Internet connection and a current browser.

6.1.2   Organization

The repository is broadly organized as three linked libraries:

- Data
- Tool
- Model

The data library holds the results of the project activities. The major component is the performance measurements on the target systems, but the library also holds documentation files for the data, discussion messages, and project reports. The tool library holds the programs and supporting files used to measure performance, analyze data, and operate the repository; both material developed by the project, and specialized tools obtained elsewhere. The model library was reserved for the details of simulated systems.

6.1.3   Interface

All three libraries are accessed through the WWW interface, with their contents interlinked. Although the material housed in the libraries can be accessed in a traditional hierarchical manner (the repository main page links to a page for each library, which links to the directories where the files are available for download), the data and tools are designed for online interactive use.

For example: The data library main page lists the different types of data available, with links to search tools that identify files of that type. Links on the search results point to "home pages" for each data set that are, in turn, generated by another tool. The home page display includes information about the data (drawn from the data library) marked with links to the raw data, and documentation for each component and to tools which can be used to analyze it. Links to the analysis tools pass the home page information; this permits the tool to immediately load the data from the library and put up an initial display of the selected variable(s), without the need to manually specify the data file or variable names.

All three libraries share a common upload procedure, to allow users to contribute material to any library. And the discussion board software permits links to any resource.

6.2   Input

The starting point for the repository is the ability to accept material from the outside. This includes the results from the other two teams, as well as (we hope) contributions from outside researchers or other interested parties. There were two major entry points:

- The file upload page
- The discussion board.

The upload page provides a standard method for contributing larger files such as those for source code, data, or documentation. The discussion board provides a simple method for exchanging comments or making experiment reports.

6.2.1   Upload

The three libraries share a common upload procedure. An HTML form linked from the repository home page allows the uploader to enter information about the file and attach the uploaded material.

The form requests all the information needed to handle the data:

> Uploader's name and e-mail address
> Contact name and e-mail address for information about the file
> Data set title
> Detailed description of the data set
> File type (data, source code, graphics, text)
> Archive library (data, tool, model, discussion images)
> Name of the file

The receiving script formats the information from the upload form to create an "info" file describing the attached object, writes both to a temporary directory, and e-mails the info file to the site maintainer. Images for the discussion board are automatically processed and added to the online archive; all other material is saved for review by repository staff.

6.2.2   Discussion

To facilitate the roles of sharing and coordinating, the repository maintained an online message board. The software used was WebBBS <http://awsd.com/scripts/webbbs/>, a simple, freely distributed, off-the-shelf product.

Configuration options allowed messages to include HTML; permitting both formatting and embedding of links and images. They also allowed the inclusion of header/footer sections to make the general layout of the pages consistent with the repository style. Users could individually select message threading and selection options, and save these in their browser with http cookies. The control software was stored in the Tool library; the messages were stored in the Data library. A special section in the Data library was established for image files to be included in discussion messages; the file upload procedure was set to allow these images to be placed directly into the library, without intervention by repository staff.

The ability to embed HTML, and upload images was used to facilitate cross-linking of resources. Files, or programs referenced in a message could be linked to the appropriate page in the repository (or to off-site locations). Repository staff were able to produce well- constructed messages which presented experimental results in a detailed and attractive manner, with embedded links to the data files referenced, programs used, relevant documentation, and previous discussion messages referred to.

It is worth noting, however, that more casual users found the process of constructing such detailed documents arduous and confusing. To achieve the desired goal of fully linked and illustrated discussion in a production system, images would need to be attached directly to messages, and cross-links inserted automatically.

6.3   Data Library

The Data Library itself consists of a directory tree on our main data server. In the repository's cross-linked structure, files may be accessed directly by http, but are designed mainly for online use through programs in the Tool library. There is a directory for performance data collected on each computer platform, a directory for documentation files, and a directory for discussion messages (with a subdirectory for images).

6.3.1   Performance Data

The repository holds both the output from the SESAME instrumentation, and data from the user-level programs used to generate system activity. User-level data is held from both the target platform of the Intel Paragon, as well as comparison platforms such as the Sun SPARCstation. In some cases, special files were created merging the SESAME data taken at the operating system level, with the data generated by the user program running at the same time.

Data sets are stored as pairs of files, one containing the data itself, the other containing information about the data. The data files are all in binary SDDF format; the "info" files are flat ASCII. The files are held in the directory for their platform in the data library, with a file name based on the host, time, and instrument; the info files are identified with the further extension "info."

Data Files

Data files are all held in the binary <u>Self-Defining Data Format</u> (SDDF) developed by the <u>Pablo</u> Project at the UIUC. This format allows both elaborate labeling of the data, and transfer of binary data across systems with different internal numeric representations. A library of C++ objects allows for decoding and manipulation. The SESAME data is written directly in this format; all other data was converted to this format with special routines using the library functions.

The file names were constructed to encode basic information. The SESAME data files were constructed from the hostname, an underscore, and a date-time string. Other data types used a suitable extension. Data taken with a file system benchmark called "iow" on the repository SPARCstation Greyhound, at 10:02:41 on 20 March 1998, has the file name <u>Greyhound_199803201 00241.iow</u>; the matching info file has the name <u>Greyhound_199803201 00241.iow.info</u>.

For detailed descriptions of the data, see the documentation files.

Info Files

The Info file contains a description of the data. It is in flat ASCII format to ensure readability, and follows a standard layout both to be comprehensible to human readers, and to allow scanning by automated tools. It includes:

Information from the upload form:
> File name
> File size
> Contact name and E-mail

Data set title
Detailed description
Original file name

Information drawn from the file:
System type
Host name
NumNodes
Instrument type and version
Begin date

And structure:
Packet structure
Packet type
Different types of data records used
Data record descriptions

6.3.2 Documentation Files

Online documentation files are provided for each major system component. They are held mainly as HTML, to be cross-linked from other parts of the repository. The Data library home page includes a short description of each type of data, with links to the documentation and data files which include that type. Each data file's home page includes a list of the data types the file contains, with links to the documentation and analysis options. And links to the documentation can be embedded in discussion messages where the data is used. Documentation includes:

- The SESAME design documents. The original MS-Word files for both the high-level conceptual description and the detailed low-level design documents are included. An HTML version of the high-level document was prepared for online viewing.

- The SESAME control GUI user instructions.

- Details for all SESAME subsystems. These provide a list of the data produced by the subsystem, and information on its interpretation and handling.

- Descriptions of the user-level programs and the data they produce. The various forms of the file system benchmark ("iow"), and the test workload generator (sesame-sort) are covered.

- Details of the disk drives attached to the test Paragon. The file systems were a significant focus of investigation, and the manufacturer's data sheet was obtained and posted in its original (ASCII) format.

A subdirectory is included for the SESAME QUORUM '98 presentation. It is presented in both the original PowerPoint files, and in carefully prepared HTML.

6.3.3 Discussion Messages

In normal operation the WebBBS files, both the program and message files, would usually be hidden from the users, and accessed only through the BBS interface. As part of the research goal, they were made directly accessible to allow users to view the repository's design. A subdirectory for images included both the image files and info files similar to those for the data automatically generated from the data on the upload page.

27

6.4   Tool Library

Just as the Data library holds the measurements and supporting documentation, the Tool library holds the programs, and supporting files used by the project. This includes both software developed by the project and resources obtained from outside. The major categories are:

- Measurement Programs
- Data Handlers
- Analysis Programs
- External Tools

6.4.1   Measurement Programs

The performance measurement software is housed in the library, both the SESAME OS level and user-level benchmarks. Reasonably complete distributions were prepared for each program, with source code, makefiles, and supporting files such as test output. The files could be downloaded by http, so that outside researchers could install and use the programs on their own systems. It was hoped that they would upload the results and improved programs to the Repository, but time and circumstance did not permit this.

SESAME

There were three versions of the portable components of the SESAME OS level software put online, along with a control GUI. The GUI was put online on a test basis so that remote users could operate the OS level software on target machines. However, routing difficulties and conflict prevention issues made this problematic.

It is worth noting that while the portable components of the SESAME system could be put online and included as part of the project deliverables, the data collection hooks themselves could not. SESAME hooks are added directly to the OS source code, literally becoming part of the operating system itself. The Paragon operating system is proprietary, and no satisfactory way was found to disclose the nature and placement of the hooks without disclosing Intel's intellectual property. Platinum concluded that they could supply instrumented kernel modules to Paragon owners, but not release the material to parties not otherwise authorized to possess it.

In addition to the SESAME programs, there were two major user-level programs developed for the project, *iow* and *sesame-sort*.

iow, File System Benchmark

The Paragon file systems were a major focus of the project. A general-purpose benchmark program was developed by the modeling team, and several specialized versions were developed by the repository staff. Each version included a format converter to produce SDDF versions of the data, and calling scripts to collect associated information about the system configuration.

sesame-sort, Workload Generator

Repository staff developed a special user program to drive the instrumented Paragon through each instrumented activity in the manner of a real parallel application. The program was a parallel

sort routine that reads from a Parallel File System (generating measurable PFS reads); sorts the input data (loading the CPU); passes sorted data between nodes with MPI (requiring transfers of varying sizes); writes to PFS (generating measurable PFS writes); and handles data sets large enough to force significant virtual memory operation (across the various measurable types).

6.4.2 Data Handlers

Several data handling routines were developed to help install data in the repository, and allow web-based interaction with the SDDF format data files.

Data Installation

As described above, data sets consist of two files: the data and an accompanying info file. The input procedure gathers most of the data for the info file. A routine called FileStructure is then used to scan the data file, and append a list of all data record types, their number, and the attributes and data types of each field. The files are then placed in the online directory.

Search

Several search functions were installed on the Web site:

- Full text of discussion messages
- Full text of documentation files
- Basic information by platform and instrumentation

The WebBBS program includes the ability to search the messages for keywords. The repository adapted the search code to work from a general search page.

A simple routine (searchDocs.pl) "grep"ed all the documentation files for a string from the search form, and returned links to all matching files. The most basic data for each file was:

- File name
- Data set title
- Platform type and name
- Instrumentation
- Creation date

This was extracted from the info file with a Perl routine (baseload.pl), and placed in an Oracle database. A Perl/SQL interface (platbyinst) called from a simple HTML form permitted users to search for data files from any platform, any instrument, or any combination.

SDDF Interface

The SDDF format is very flexible, but quite complicated. It can only be read effectively with the specialized library routines in the Pablo distribution. To convert data to more easily readable forms, the repository used the "FileStructure" program to extract descriptive information, and adapted the Pablo "ExtractFieldValues" to retrieve data. The adapted program (S-ExtractFieldValues) improved the numerical accuracy of the output, permitted extraction of

strings and extraction of arrays in several output formats. Perl routines could then scan the info files for descriptions, and call the extractor to read the associated data.

6.4.3 Analysis Programs

Online analysis of data was a major focus of the repository, and a number of cross-linked routines were developed. As with the data generation programs, the source code and supporting files were made available for download, but the emphasis was on use through the WWW. There were two display tracks: quick-look displays which generated bitmaps using Perl; and Java-based displays which were more sophisticated.

File Home Page

The primary entry point for analysis is the data file's home page. On demand, a Perl routine scans the info file, and generates a Web page showing the data record types, the analysis options, and the file description (Figure 6.1). The data type listings link to the documentation, and the analysis option links invoke each against the file.

Figure 6.1: Data File Home Page
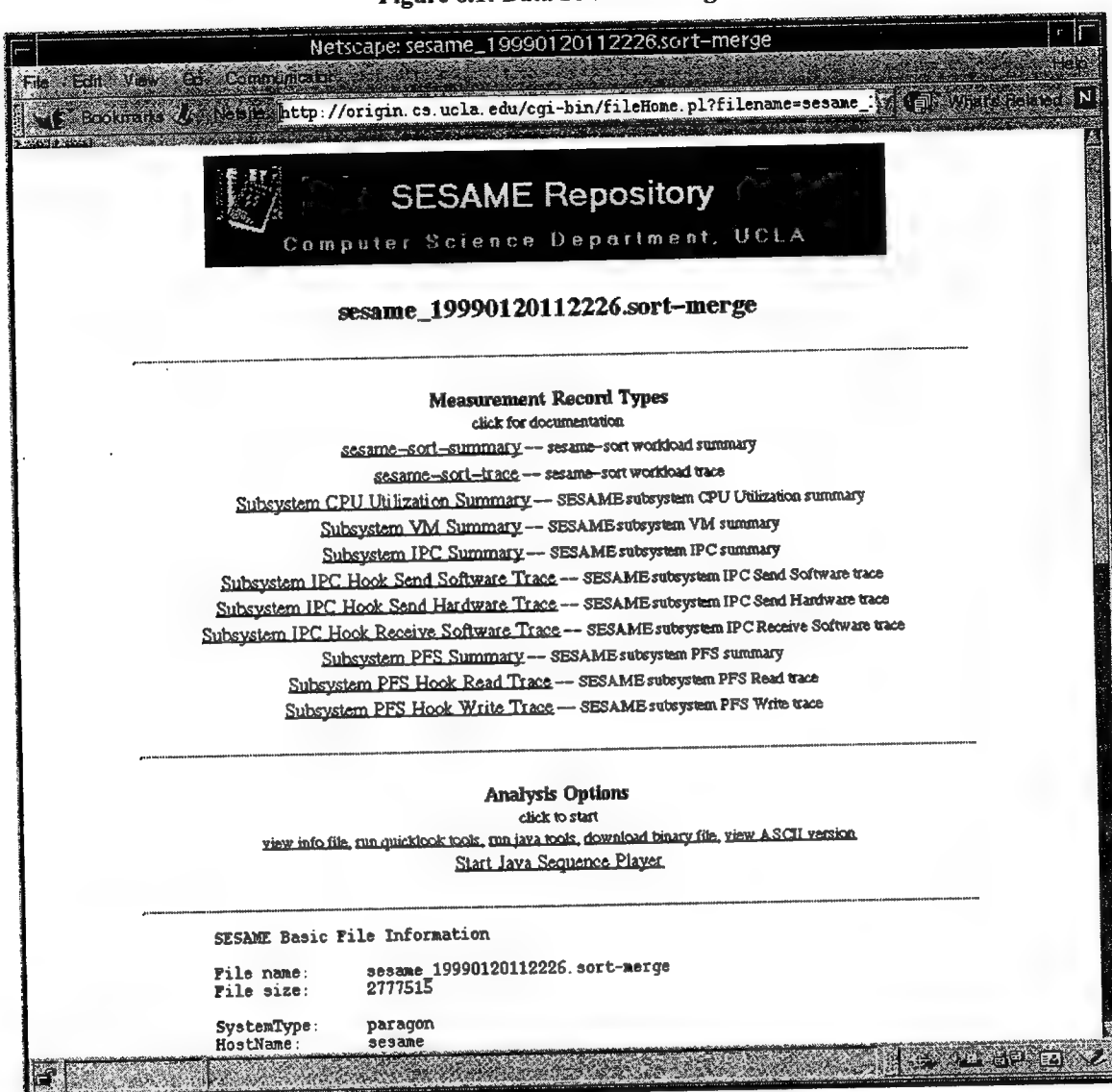
Quick-Look Tools

The performance measurement tools produce large quantities of data whose properties are not well known in advance. This creates a need for a fast, simple system to check the results without the effort of extracting the data and entering it into a plotting program.

The quick-look tools are highly automated routines, which produce bitmapped plots of selected data. The basic design generates a standard set of labels from the info file, and autoscales the data to produce the initial view with minimal input. Different routines adjust the details for the target data, and usually present the plot in an HTML form to allow customizing the maximum, minimum, and scaling.

The minimal information required for the initial view allowed the process of invoking it to be highly streamlined. A start page linked as "run quicklook tools" from the file's homepage

scanned the info file for the available data, and generated links and forms showing the applicable plot types. Clicking a link, or submitting the form would proceed directly to the initial view, without the need to manually enter any of the instructions for the plot. Four types of plots were developed:

- Count/rate
- Bivariate
- File record IO time
- Sort mixed-level.

SESAME summary records include the counts of each measured event over the pull interval. For a selected field, the count/rate routine produced a plot with a small graph for each node, showing the count, and count rate over time (countRate.pl). The plot was presented as an image map, so that clicking the graph for a node generated a larger, rescaled plot of the data for the node.

For scalar data, a simple form allowed the user to plot pairs of fields within a record (showXY.pl).

The file-system benchmark could measure the I/O time of each record. A special version plotted this data (mt2.pl).

The workload generator produced matching data from both the user and system levels. Another quick-look routine combined much of the data to show how CPU loading and virtual memory activity varied with the applications actions (showSequence.pl).

As an example, Figure 6.2 shows a section of the quick-look page for a file-system benchmark file. Selecting the link marked "Plot IO Timings" brings up Figure 6.3 listing the data for each run contained in the file. Selecting the link for the run with an "sio" (size of record) of 68K produces the plot shown in Figure 6.4 displayed with a form for adjusting the plot parameters.
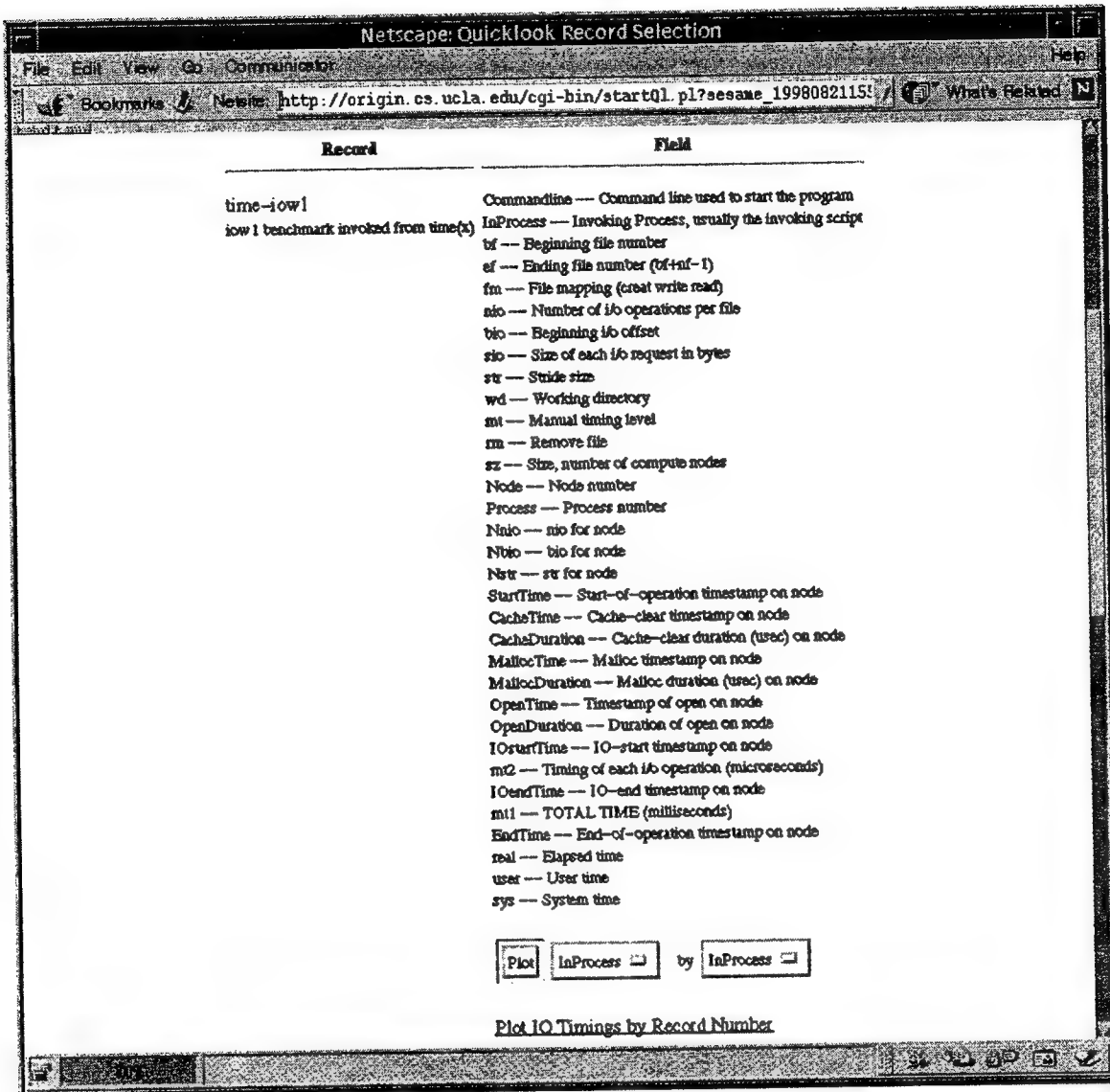
Figure 6.2: Section of a Quick-look Tools Page

Netscape: Quicklook Record Selection

File   Edit   View   Go   Communicator   Help

Bookmarks   Netsite: http://origin.cs.ucla.edu/cgi-bin/startQl.pl?sesame_1998082115!   What's Related

Record	Field
time-iow1	Commandline --- Command line used to start the program
iow1 benchmark invoked from time(x)	InProcess --- Invoking Process, usually the invoking script
	bf -- Beginning file number
	ef --- Ending file number (bf+nf-1)
	fm --- File mapping (creat write read)
	nio --- Number of i/o operations per file
	bio --- Beginning i/o offset
	sio --- Size of each i/o request in bytes
	str --- Stride size
	wd --- Working directory
	mt --- Manual timing level
	rm --- Remove file
	sz --- Size, number of compute nodes
	Node --- Node number
	Process --- Process number
	Nnio --- nio for node
	Nbio --- bio for node
	Nstr --- str for node
	StartTime --- Start-of-operation timestamp on node
	CacheTime --- Cache-clear timestamp on node
	CacheDuration --- Cache-clear duration (usec) on node
	MallocTime --- Malloc timestamp on node
	MallocDuration --- Malloc duration (usec) on node
	OpenTime --- Timestamp of open on node
	OpenDuration --- Duration of open on node
	IOstartTime --- IO-start timestamp on node
	mt2 --- Timing of each i/o operation (microseconds)
	IOendTime --- IO-end timestamp on node
	mt1 --- TOTAL TIME (milliseconds)
	EndTime --- End-of-operation timestamp on node
	real --- Elapsed time
	user --- User time
	sys --- System time

[Plot] [InProcess ▼] by [InProcess ▼]

Plot IO Timings by Record Number

Figure 6.3: Top of Selection Page for Benchmark I/O Time Display

File Edit View Go Communicator

Bookmarks Notes: http://origin.cs.ucla.edu/cgi-bin/mt2.pl?filename=sesame_19980... | What's Related

SESAME Repository
Computer Science Department, UCLA

SESAME Quick Look — IO Time by Record Number

There are 64 **time-iow1** records in sesame_19980821155550.iow1.
64 have io timings.

Select the link for the file number and run.

Parameters

Run Number	nio	sio	str	wd	File 0 mode
0	256	4 KB	0	/pfs/iow/	creat
1	256	8 KB	0	/pfs/iow/	creat
2	256	12 KB	0	/pfs/iow/	creat
3	256	16 KB	0	/pfs/iow/	creat
4	256	20 KB	0	/pfs/iow/	creat
5	256	24 KB	0	/pfs/iow/	creat
6	256	28 KB	0	/pfs/iow/	creat
7	256	32 KB	0	/pfs/iow/	creat
8	256	36 KB	0	/pfs/iow/	creat
9	256	40 KB	0	/pfs/iow/	creat
10	256	44 KB	0	/pfs/iow/	creat
11	256	48 KB	0	/pfs/iow/	creat
12	256	52 KB	0	/pfs/iow/	creat
13	256	56 KB	0	/pfs/iow/	creat
14	256	60 KB	0	/pfs/iow/	creat
15	256	64 KB	0	/pfs/iow/	creat
16	256	68 KB	0	/pfs/iow/	creat

34

**Figure 6.4: Quick-look Plot with Adjustment Form**
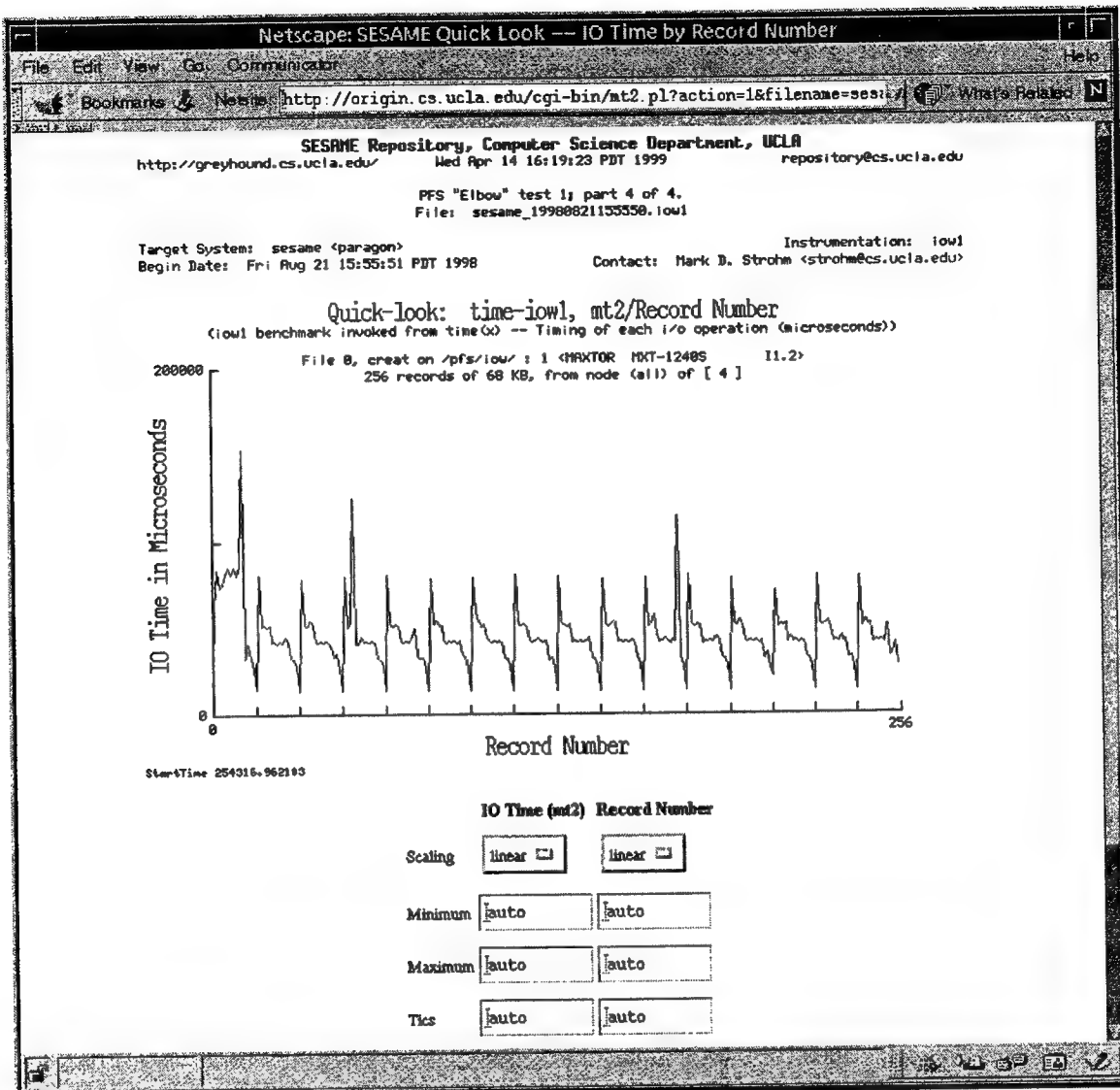
Java Tools

The Java tools were intended to allow production of better-looking, and more customized displays than the quick-look tools could provide. A line graph building program was developed for scalar data (sesameApplet.java).

A special display was developed for the sesame-sort data. This "sequence player" produced a moving "play back" of the data from the sesame-sort workload generator on a diagram of the computer (SequencePlayer). For each node, bars showed the level of CPU consumption and virtual memory activity from the SESAME summary records, and lights showed inter-processor communication and PFS I/O at both the user and system levels. The application phase was also shown.

An example is shown in Figure 6.5 (the program is explained in more detail Section 6.4.4). The application is sorting 65MB of random numbers with eleven compute nodes. Nodes 4, 9, and 10

are actively sorting the numbers they currently hold; all show high CPU consumption by the user, but they show varying levels of system CPU and VM activity. Nodes 5, 6, 7, 8, 11, and 15 are each waiting to exchange numbers with the node below; they also show high CPU consumption by the user, with varying levels of system CPU and VM activity. Nodes 12 and 14 are exchanging sorted numbers; node 14 is in the hardware phase of an IPC send.

Figure 6.5: Display of Paragon Running "sesame-sort"
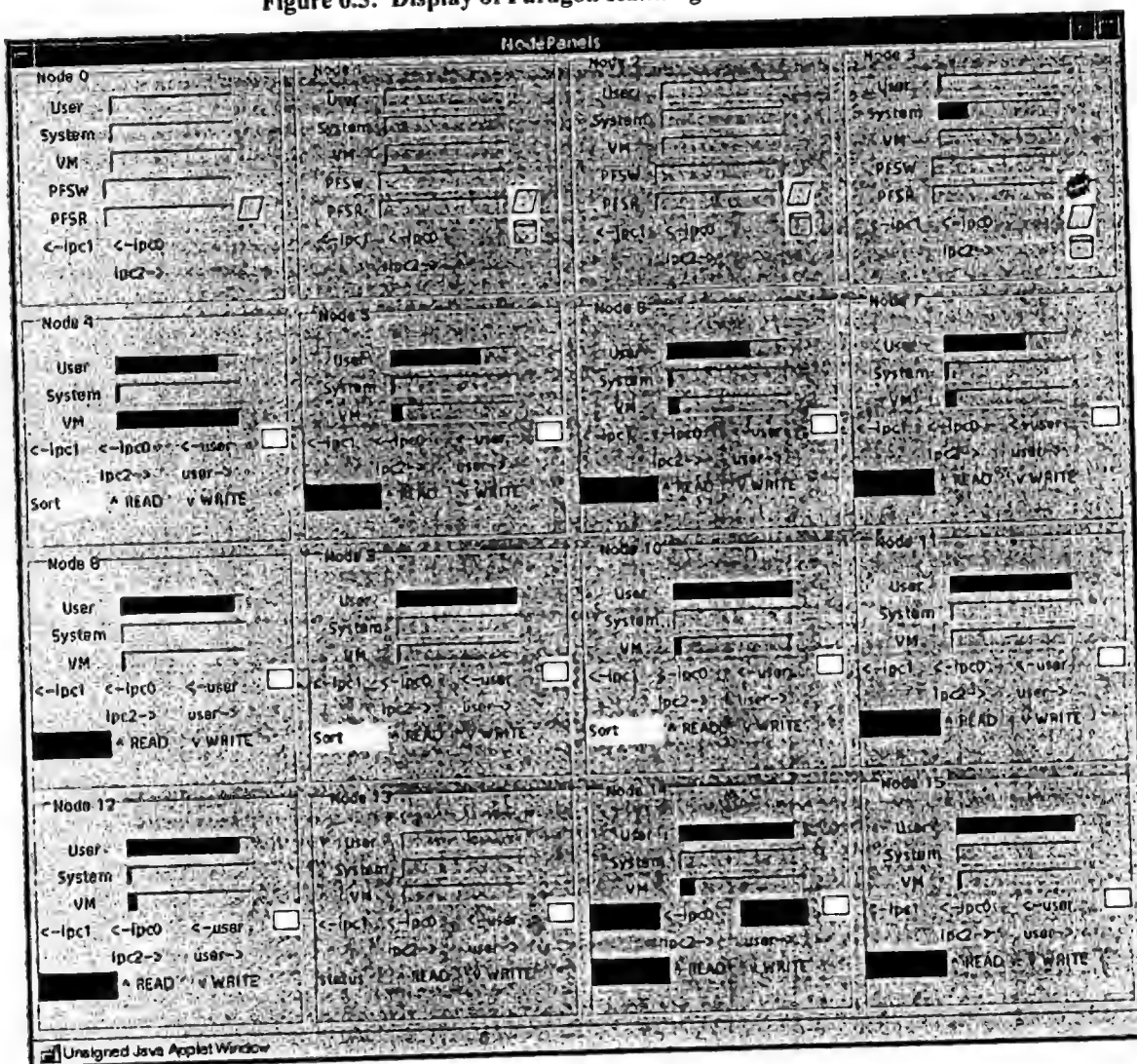
6.4.4 External Tools

There were two major external resources used: distributions from the Pablo project at UIUC, especially the SDDF handling functions, and WebBBS.

These were placed online in the Tool library, but were intended for use by other routines rather than for download by users. As with the WebBBS data files, in typical operation the config files would likely have been hidden for security reasons. Here they were exposed as part of the research goal.

6.5   Model Library

This was intended to hold different models of the target systems. These models were to be used to verify our understanding of the systems, and predict performance under speculative conditions. The library was used to store the source code for *Platinum technology's* Paragon port of the MPP simulator developed by the Modeling Team.

7. MEASUREMENT RESULTS

The SESAME performance measurement system is intended for use on fully running systems with actual applications whose performance one wishes to improve. If a program's performance is inadequate in an OS related area, investigators can instrument the relevant section of the OS code, run the target application, and produce a detailed picture of the process. For our target MPP systems this was expected to reveal suspected inefficiencies in the OS itself (and did), but the insight gained could also be used to tune an application to the system.

7.1 Test Program

To demonstrate our ability to gain this insight, repository staff primarily used a test program called "sesame-sort." It is designed to generate system activity similar to an actual parallel application (driving the computer through each of the major instrumented activities), but be simple enough to follow its behavior with a minimum of tortured analysis. The program is an inefficient parallel sort, whose task is simply to read a portion of a large file of random double precision numbers, sort them, and write them to a different file.

Full details are given in the documentation, but the basic algorithm is this:

> Each compute node reads a section of the random number file and sorts it. The node then tests with its neighbors to see if their sorted lists overlap. If so, they exchange a portion of their highest and lowest numbers, and resort. The exchanges and resorts continue until the nodes find no more overlap in the lists. The input numbers are now fully sorted, and the nodes write their lists to the output file.

The files are held on a parallel file system, so that reads and writes trigger the SESAME-PFS instrumentation. Trades are done using message passing interface calls, which are captured with the inter-processor-communication (IPC) instrumentation. Sorts use a standard heap-sort algorithm, which both generates a high processor load measured by the SESAME-CPU instrumentation, and sweeps through the allocated memory producing virtual memory activity measured by the SESAME-VM instrumentation.

The program issues timestamps as it passes from one activity to another, which permits us to correlate the user-level activity with the SESAME system-level data. A calling script invokes the program through a timer, to record execution time and CPU consumption data normally available at the user level.

7.1.1 Computer

The computer used for the tests was a 16-node Intel Paragon XP/S. It is diagrammed in Figure 7.1. It is partitioned into service (I/O) nodes (0-3), and compute nodes (4-15); node 3 is the boot node. The I/O nodes communicate to the attached devices, and run basic user processes; the compute nodes run the parallel programs. Nodes 1 through 3 each have a 1GB disk drive attached; the drives on 1 and 2 hold a parallel file system; the drive on node 3 holds an unused PFS section, and all other file systems. Due to a parity error on the backplane, the slot for node 13 is not usable, and no node is present.

The system uses "GP" nodes, with two Intel i860, processors; one for computation, and another for communication. The compute nodes have 16MB of RAM, the I/O nodes have 32MB.

Figure 7.1: Test Computer
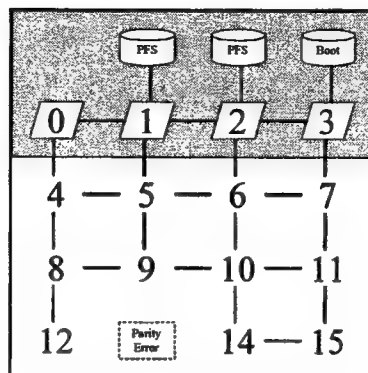
7.1.2 Tests

Runs were made with all seven possible numbers of compute nodes (Table 7-1) Sort sizes varied from 1MB to 65MB.

Table 7-1: Combinations of Compute Nodes

Size	Nodes
1	4
2	4 8
3	4 8 12
4	4 5 8 9
6	6 7 10 11 14 15
8	4 5 6 7 8 9 10 11
11	All

Our analysis will focus on the data obtained with the SESAME instrumentation, rather than the application. But a general view of the program's performance is given in and Figure 7.2 and Figure 7.3. These graph the wall clock time required for the runs to complete, at small run sizes and over the full range, respectively.

At smaller sizes we can see a general overhead associated with larger numbers of compute nodes, but a slower increase in completion time as the sort size increases. With one compute node the times follow roughly the $N\log_2 N$ scaling of the underlying heapsort. The other curves rise more slowly; by about 6MB two nodes complete the task faster than one; at 10MB the best performance is with three.

At larger sizes, however, performance is dominated by abrupt increases in time when the sort size reaches between 8 and 10 MB per node. Interestingly, a sort of 65MB with 8 nodes causes the system to panic on a VM paging error.

39

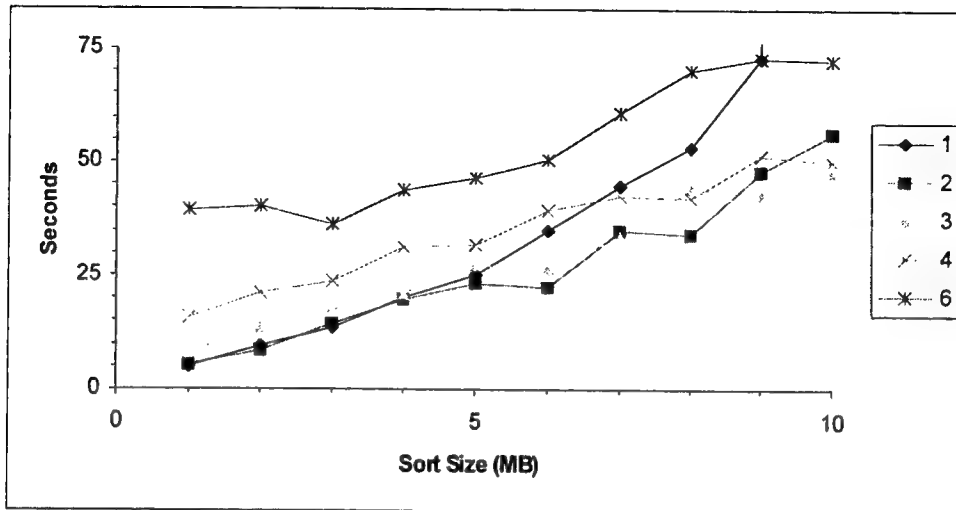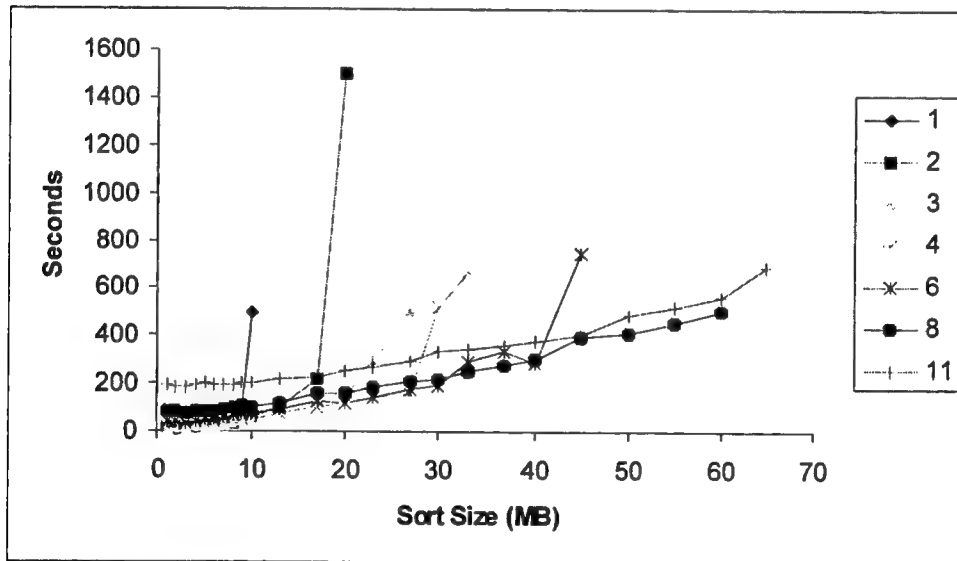**Figure 7.2: Run Completion Times with Small Sort Sizes**

Figure 7.3: Run Completion Times, Full Range

7.2   MPIinit, High Time Resolution

The SESAME Inter-Processor Communication hooks detected an unexpected burst of MPI activity between the start of execution and the allocation of the main data array on each node. From the timing and distribution of events, it appears that this is the MPIinit procedure, which is called on each compute node to initialize the communication structure. These data indicate a potential scaling problem in the procedure, and demonstrate the extreme detail the SESAME instrumentation can reveal.

7.2.1   Init Phases

In the cases examined these bursts have four phases:

- Each node sends a zero length message to each other node.
- Zero length messages are passed hierarchically from higher ranked nodes down to the zero ranked node, which then returns a message along the same path.
- Zero length messages are again passed hierarchically, but by different paths..
- Phase III repeats with a message size of 4 bytes.

Nodes move from Phase I to II immediately, so that the two phases overlap in time. There is an ~0.1 second delay between Phases II and III, and III and IV. The content of the messages is not known, nor is the underlying algorithm.

Figure 7.4 shows the pattern of communication in each phase during a four-node run in data file sesame_19990119124421.sort-merge. In Phase II, node 9 exchanges messages with node 5, while in II and IV it exchanges with 8. In the path phases nodes wait until receiving all the messages from above before sending to the next in line; in Phase II node 5 sends to 4 after receiving from 9, and 4 sends to 5 and 8 after receiving from both (more details follow below).

Figure 7.4: MPIinit Phases Observed with SESAME
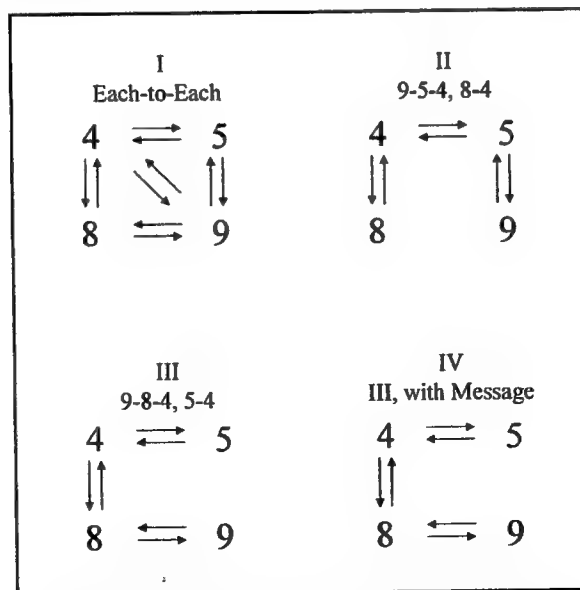
7.2.2 Scaling Issue

There are potential performance problems in the scaling of this procedure. The each-to-each phase scales as N(N-1); the largest Paragons would have supported about 2000 compute nodes for around 4,000,000 nearly simultaneous messages, many between distant points in the mesh. The other phases might suffer from complexity in their scaling, as each node must know which nodes will be passing to it, and wait until it has received from all before sending to the next node in the path.

7.2.3 SESAME Trace

The SESAME instrumentation allows us to trace this activity in great detail, using actual measurements rather than assumptions, extrapolations, or estimates. We have traced the pattern of execution in several cases, and detected at least one performance anomaly.

As an example of the tracing detail, let us consider Phase II above. Each message is measured in three places. The transfer from the user level through the OS kernel to the hardware; the hardware transfer itself; and the pass through the receiving node OS to its user level. Each measurement gives the time of occurrence and duration of the event, allowing us to trace the process. Figure 7.5 shows a diagram of Phase II with letters showing the order in which the messages were passed. Table 7-2 and Figure 7.6 show the data to microsecond accuracy.

The level of detail achieved is very high; the graph is only 500 microseconds wide. We can see, for example, that Node 5 was ready to receive the message from 9 immediately (1μsec after hardware transfer), but 4 does not begin its receive from 8 for 200μsec. But this delay added less than 100μsec to the phase completion (the delay between the 5→4 send and receive) because 8 would still have had to wait for the 9→5→4 path to complete.

Given that these, again, are measures of actual execution, an error or inefficiency in the algorithm could hardly escape such inspection.
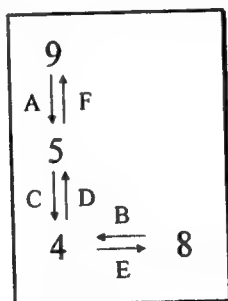
Figure 7.5: Phase II Diagram

Table 7-2: Data from Example Phase II

| Event	Send Software Start	Send Software Stop	Send Hardware Start	Send Hardware Stop	From Node	To Node	Receive Software Start	Receive Software Stop
A	91675.618190	91675.618201	91675.618215	91675.618220	8	4	91675.618420	91675.618431
B	91675.618275	91675.618289	91675.618301	91675.618307	9	5	91675.618308	91675.618318
C	91675.618344	91675.618359	91675.618371	91675.618377	5	4	91675.618475	91675.618483
D	91675.618523	91675.618533	91675.618551	91675.618557	4	5	91675.618557	91675.618567
E	91675.618575	91675.618583	91675.618596	91675.618602	4	8	91675.618606	91675.618623
F	91675.618589	91675.618599	91675.618616	91675.618621	5	9	91675.618624	91675.618636

Figure 7.6: Phase II Graph
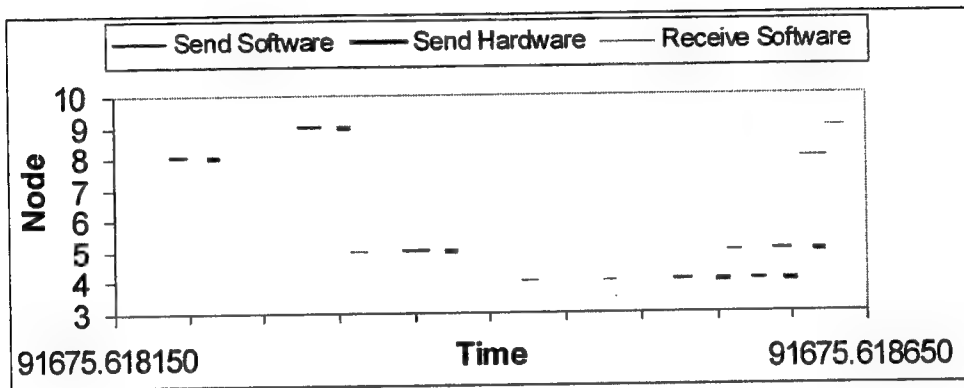
7.2.4 Scaling Test / Performance Anomaly

Without a knowledge of the algorithm, the shapes of the passing networks in large systems cannot be predicted. But for comparison, Figure 7.7 shows the patterns for the largest array we could produce: 11 nodes (from the same data file as above). Phase II shows a pattern based on pairs. Phase III shows a chain-oriented layout. In the 4-node case, a node was at most only two hops out from the center, in the 11-node case it is three or four hops out.

Figure 7.7: Phases II and III with 11 Nodes
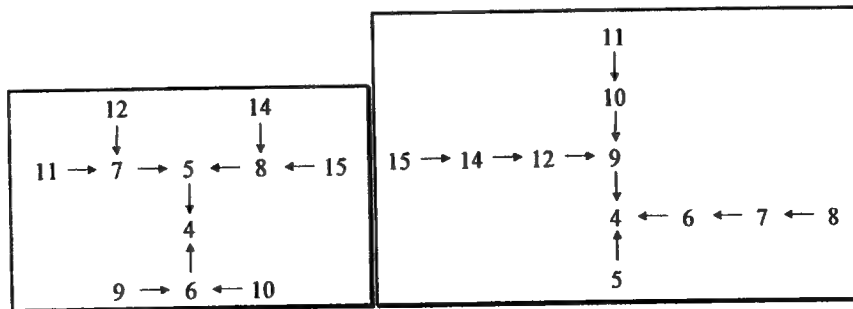
Although the test computer was too small to investigate massive scaling effects, we would still like to check the scaling issues. Because the nodes proceeded separately, and the phases were

43

short, this is difficult. Phase IV is unusually discreet and identifiable and we attempted a measurement by checking the time from the start of the earliest *send software*, to the complete of the latest *receive software*, and compared this to the number of nodes. The results are shown in Figure 7.8.

As feared, there is no visible scaling. But there is a repeated anomaly. For six compute nodes, the times are bimodal, concentrated both at the typical time of around 6 milliseconds, and at a much higher value of 20. By tracing the execution, as in the example above, we found that this was due to an unusually long interval between the *send* from node 14, to the *receive* by node 11. The *send* and *receive* events themselves are not of unusual duration, but the interval between them is extended.

The significance of this is that timing anomalies do occur in the MPIinit function, and that SESAME instrumentation can describe them in great detail.

Figure 7.8: Phase IV Completion Times
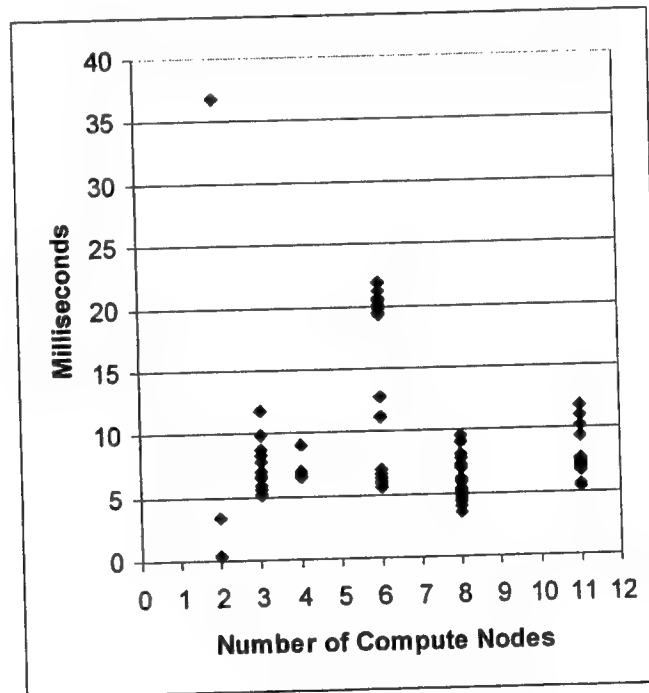
7.3   Parallel File Open, Coordination of Measurements

The test application read its data from a single input file, requiring that the nodes coordinate their reads so that each acquired a separate section. The Paragon OS provided a good means for this with a mode in which the system automatically determined the location to read based on the number of compute nodes and the record size used by the application (M_RECORD). The process of opening and synchronizing the input file, however, turned out to be much longer and more complicated than expected. By coordinating the data from the application and multiple SESAME hooks, we can obtain some insight into the procedure, and why it takes as long as it does.

### 7.3.1   Time to Open and Synchronize a File, from Application

From the application level we can observe the amount of time it takes to open and synchronize the file. On each node, the program issues timestamps immediately before the commands to open the file, set the mode, and read the data. No other actions are taken during the sequence. By comparing the stamp for the open command from a given node with the stamp for the read, we can determine the time it required for the open/sync procedure. (Because the nodes execute separately, it is easier to define this per node, rather than attempt to compute a global number.) Results are shown in Table 7-3 and Figure 7.9. The mean time to open and sync scales very linearly, about 0.14 seconds per node.

For a scalable system, this is a substantial number. In a 2000-node machine, it would imply an open time of over 4 minutes. If the run is to last many hours, this might not be a problem, but it becomes serious if short execution is required, or many files are used. For an 8-hour run, opening ten files would consume 10% of its allotment. An "embarrassingly parallel" procedure that takes a day to run on one node, might be expected to require only a minute on 2000, but opening only one input file and one output file would extend the amount to ten times this value.

Table 7-3: Time to Open and Synchronize Input File

Number of Compute Nodes	Event Count	Mean Completion Time	Scaling
1	10	0.101	0.101
2	26	0.291	0.145
3	45	0.459	0.153
4	68	0.564	0.141
6	114	0.794	0.132
8	184	1.110	0.139
11	264	1.507	0.137

45

Figure 7.9: Mean File Open Time and Scaling
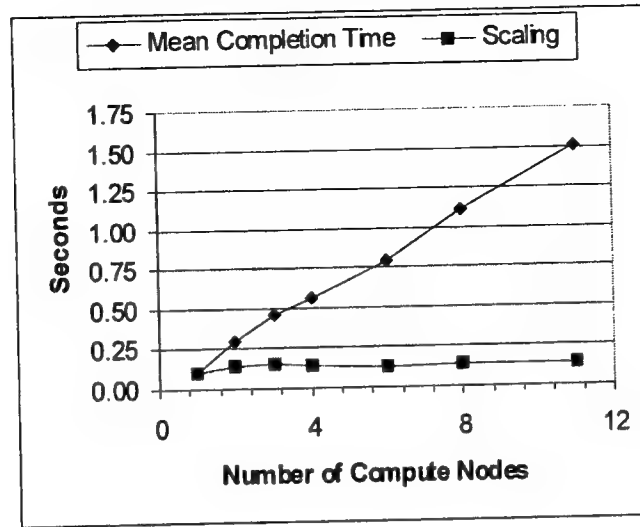
7.3.2 Coordinated Tracing

The application-level numbers can be used to locate the problem and indicate its magnitude, but they provide no insight into its source. By combining the application's timestamps with SESAME trace data, we can shed much more light on what is taking so long.

On each node, the application allocates an array to hold the input data, opens the input file, sets the file mode (which requires a global synchronization), and then reads the data into the array; printing a timestamp immediately before each action. These actions follow one another directly, with no other I/O or significant computation.

Table 7-4 shows the trace for these events and the SESAME PFS and IPC hooks, for a run using two compute nodes (from file sesame_199901119124421.sort-merge). The "Time" column shows the timestamp of each application event, and the start and stop times for PFS reads (READ), IPC software receives (RECV), and IPC hardware and software sends (SEND). The "Size" column shows the number of bytes read from the file or transferred by MPI.

The SESAME instrumentation shows that each open command is followed immediately by a PFS read on Node 3 (the boot node), and the sync is accompanied by an exchange of MPI messages between the compute nodes. The requested size of the PFS read was 8192 bytes (one virtual memory page) but the data returned was only 104 bytes. The IPC messages were in a series of sizes: 24, 28, 100 and 0 bytes.

From the timestamps, it is clearly visible that the IPC messages are performing the file synchronization, exchanging messages of different sizes. With larger numbers of compute nodes the situation is more complex, with the nodes exchanging more messages, the messages having a wider array of sizes. The nodes also proceed through the sync procedure slightly out of phase, such that messages of different sizes overlap, and some nodes begin reading from the file while others are still performing the sync. But this is clearly the primary use of the time.

Without knowing the algorithm or exact requirements for the synchronization, it is difficult to say, but this would appear to be excessive. Computing the actual file locations used by each

46

compute node requires knowing only the number of compute nodes, the rank of the node in question, and the record size. It should be possible to sort this out among the nodes with far less communication.

Table 7-4: File Open/Sync Trace

Node	Event	Size	Time			
4	array		91620.145266			
4	open		91620.146660			
3	READ	8192 (104)	91620.190408 -	91620.202103		
8	array		91620.190619			
8	open		91620.191462			
3	READ	8192 (104)	91620.226979 -	91620.230664		
4	sync		91620.251981			
8	sync		91620.276960			
4	SEND	24	91620.318820 -	91620.318851 ,	91620.318886 -	91620.318908
8	SEND	24	91620.346989 -	91620.347015 ,	91620.347040 -	91620.347064
4	RECV	24	91620.347068 -	91620.347108		
8	RECV	24	91620.347108 -	91620.347122		
8	SEND	28	91620.358300 -	91620.358315 ,	91620.358338 -	91620.358361
4	SEND	28	91620.369034 -	91620.369051 ,	91620.369076 -	91620.369094
8	RECV	28	91620.369087 -	91620.369119		
4	RECV	28	91620.369139 -	91620.369155		
8	SEND	100	91620.369246 -	91620.369260 ,	91620.369274 -	91620.369305
4	SEND	100	91620.369347 -	91620.369360 ,	91620.369376 -	91620.369397
8	RECV	100	91620.369398 -	91620.369420		
4	RECV	100	91620.369425 -	91620.369437		
8	SEND	0	91620.417613 -	91620.417627 ,	91620.417650 -	91620.417660
4	RECV	0	91620.435826 -	91620.435837		
4	SEND	0	91620.435935 -	91620.435952 ,	91620.435978 -	91620.435989
8	RECV	0	91620.435988 -	91620.436024		
8	SEND	0	91620.436332 -	91620.436344 ,	91620.436369 -	91620.436376
4	RECV	0	91620.436374 -	91620.436394		
4	SEND	0	91620.436430 -	91620.436442 ,	91620.436455 -	91620.436463
8	RECV	0	91620.436464 -	91620.436476		
4	read	523776	91620.436483			
8	read	523776	91620.436496			

7.4 PFS I/O, Possible System Errors

Engineering folklore identified I/O operations as a major site of low MPP performance. In the Paragon, the Parallel File System (PFS) was the focus of high-speed I/O, making it a prime target for investigation.

In principle, parallel disk I/O is a simple, scalable means to high speed file access. By spreading a file's contents across multiple disks, a system should be able to obtain the combined throughput of all the disks, and scale the I/O rate to required levels. The Paragon PFS proved to be a good target for SESAME, revealing it to be a complex and problematic system, with unexpected properties and inadequate documentation.

The Paragon PFS is laid out as "stripes" of 64KB per disk. Such that when mounted on two disks, the first 64KB of the file system resided on the first disk, the second 64KB on the second disk, the third on the first disk again, and so on for the full size of the system. Tests were run with systems of one, two, and three disks (and one comparison run on a larger computer with four five-disk raids). For the sesame-sort tests, the PFS was mounted on two dedicated disks connected to Nodes 1 and 2. This was to isolate the PFS operations from the other system activity on the boot node (Node 3), to which the remaining disk was attached.
Unexpected Asymmetries

7.4.1 Unexpected Asymmetries

Table 7-5 shows the PFS I/O for a sesame-sort run using one compute node (from file sesame_19990120103157.sort-merge). The application timestamps show the file open, sync, read/write, and file close events. The SESAME PFS hooks show the activity on the I/O nodes for that time. As discussed in the previous section, the file open is followed by a small read on Node 3 (the boot node).

At the read event, the application executes a single "read" statement, requesting 7332864 bytes (~7MB) from the PFS. The subsequent PFSread traces show the data being retrieved from the two I/O nodes. The "inclose" stamp indicates that the read statement has returned successfully, and the program is proceeding.

For the application, the read is an atomic event. The SESAME hooks show us that at the system level it executes as a series of eight reads, four on each I/O node. These reads occurred in pairs; the first three reading 1MB from each node, the last reading 524288 bytes from Node 2, and 517120 from Node 1.

Surprisingly, the SESAME data also shows that the reads on Node 1 took roughly twice as long to complete as those on Node 2. The duration of the first PFSread on Node 2 is 0.38 seconds, while the corresponding duration on Node 1 was 0.75. Since the nodes and disks are believed to be identical and executing independently, this is unexpected. And since all the data must be received before the application-level read can return, it doubles the time spent on that operation.

Equally surprising is that the durations of the PFSwrites are essentially the same on each I/O node. The sequence of events for the write is the same as for the read. But the first PFSwrite lasted 0.31 second on Node 2, and only 0.33 on Node 1. Since nothing had changed in the disk or I/O node configuration between the read and write events, this too is unexpected.

48

This seemingly simple task thus leaves us with two asymmetries: between the nodes on read, and between reading and writing on Node 1.

Table 7-5: PFS Read/Write of ~7MB by One Compute Node

Event	Start Time	Stop Time	Node	Bytes	Duration
inopen	170044.906952		4		0.077426
PFSread	170044.936988	170044.956245	3	104	0.019257
inpsync	170044.984378		4		0.014765
read	170044.999143		4	7332864	2.766347
PFSread	170045.005959	170045.389667	2	1048576	0.383708
PFSread	170045.007081	170045.761310	1	1048576	0.754229
PFSread	170045.791384	170046.118275	2	1048576	0.326891
PFSread	170045.792702	170046.538057	1	1048576	0.745355
PFSread	170046.568275	170046.870184	2	1048576	0.301909
PFSread	170046.568826	170047.314802	1	1048576	0.745975
PFSread	170047.345380	170047.490016	2	524288	0.144636
PFSread	170047.345727	170047.714273	1	517120	0.368546
inclose	170047.765490		4		0.042243
outopen	170085.112711		4		0.214507
PFSread	170085.159168	170085.168067	3	112	0.008899
outpsync	170085.327218		4		0.047872
write	170085.375090		4	7332864	1.600534
PFSwrite	170085.453173	170085.764087	2	1048576	0.310913
PFSwrite	170085.445139	170085.775759	1	1048576	0.330620
PFSwrite	170085.837724	170086.194476	2	1048576	0.356753
PFSwrite	170085.826115	170086.272664	1	1048576	0.446549
PFSwrite	170086.357334	170086.652929	2	1048576	0.295595
PFSwrite	170086.351555	170086.721610	1	1048576	0.370055
PFSwrite	170086.784622	170086.911961	2	517120	0.127339
PFSwrite	170086.778432	170086.946128	1	524288	0.167696
Outclose	170086.975624		4		0.121052

7.4.2 Transfer Rates

Transfer rate is a common metric for file system performance that we have used for several analyses. Our stating point is the disk drives' rated transfer speed of 22.7 to 44.7 Megabits/second (2.8 to 5.6MBytes/s). With two disks, we would expect a rate of double this, approximately 5 to 11MB/s.

Measured at the application level, the example above shows 7332864 bytes being read in 2.766 seconds (the time between the read and inclose timestamps), for an observed effective rate of 2.528MB/s. This is roughly what we would expect for only a single drive. The write moves the same quantity of data in only 1.601 seconds, for an observed rate of 4.369MB/s.

The SESAME measurements can be used to refine these values in two ways:

- First, by noting that the low rate for reads is due to the poor performance on Node 1. Its time to read 1MB is close to 0.75 seconds, for a rate of only 1.33MB/s. Node 2, with time around 0.33 seconds, shows a rate of 3MB/s; within the reasonable range for the drive speed.

- Second, by separating the time on the I/O nodes from that higher in the system. Examining the write, for example, we can partition the interval between the application timestamps into "start latency," "I/O time," "inter-I/O latency," and "stop latency." There is 0.070 seconds between the write timestamp and the start of the earliest PFSwrite, a total of 1.315 seconds between the start of each PFSwrite pair and its finish, a total of 0.186 seconds between the finish of each pair and the start of the next, and 0.029 between the end of the last pair and the outclose timestamp. This gives a total of 1.315 seconds attributable to the I/O per se, and 0.286 seconds to other parts of the operation.

7.4.3   Multiple Compute Nodes

Actual parallel applications are likely to make their file system requests from multiple compute nodes, and the file access mode used by sesame-sort is specifically designed for this.

Application Level

Use of multiple compute nodes, however, complicates the definition of effective transfer rate at the application level. For sesame-sort the average throughput per compute node tends to drop with increased numbers, because the nodes make their I/O requests nearly simultaneously, dividing the raw capacity of the file system between them. And because sesame-sort requires the nodes to coordinate, but not in a fixed way, the effects of this delay on overall performance are difficult to gauge. But a reasonable measure may be obtained by dividing the total amount of data transferred by the interval between the earliest read/write timestamp and the latest inclose/outclose timestamp.

Figure 7.10 shows the application-level read-rates across the range tested with sesame-sort. Up to 30MB the pattern is remarkably organized and stable, with a consistent rate of about 2.6MB/s. Curiously, at 33MB this pattern breaks down. There is considerably more variation, and the peak rate abruptly rises to about 5.5MB/s.

Figure 7.11 shows the write-rates. The write-rate varies violently, but appears to be clustered around 4MB/s, indicating reasonable performance on both I/O nodes.

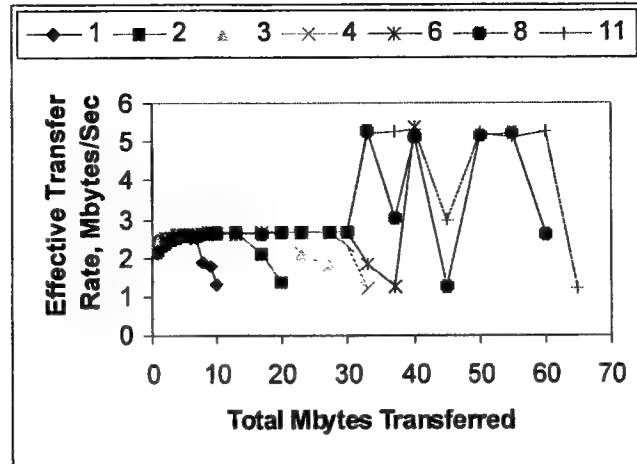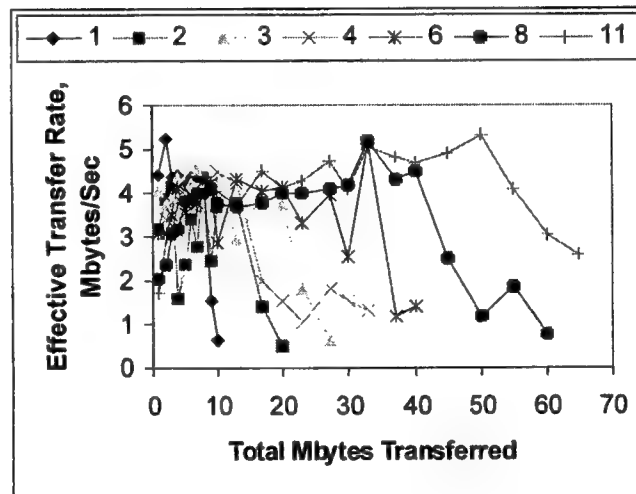**Figure 7.10: Application-Level Read-Rates by Size and Number of Compute Nodes**

Figure 7.11: Application-Level Write-Rates by Size and Number of Compute Nodes

System Level

The change in read performance above 30MB appears to be due to improvement on Node 1.

Figure 7.12 shows the PFSreads for eight compute nodes reading ~30MB (file sesame_19990125095648.sort-merge). On the I/O nodes, the data are moved with a total of 32 reads, 4 per compute node, 16 per I/O node. The graph shows each as a line running between the start and stop time for the event (the lines are positioned vertically by the order in which the event finished). The reads show an odd "nested" pattern on each I/O node, and Node 1 takes roughly twice as long to complete the sequence as Node 2. Interestingly, the time between request completions appears to be roughly stable on both nodes; again, roughly 0.75 seconds on Node 1, and 0.33 on Node 2.

51

Figure 7.13 shows the same display with eight compute nodes reading ~40MB (file sesame_199901271I0726.sort-merge). Again we can see the nested pattern, but this time the completion rates are similar for both I/O nodes (about 0.33 seconds per request), and the total rate is roughly double that for 30MB.

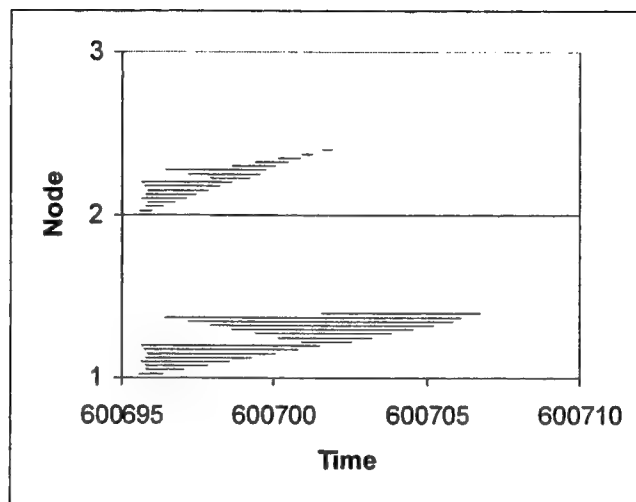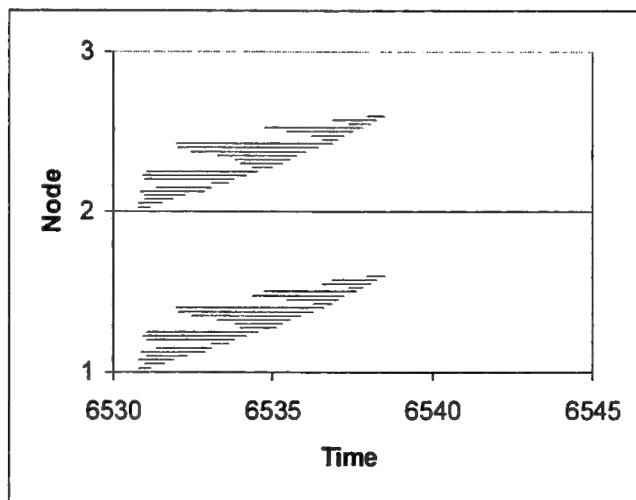**Figure 7.12: PFSread Trace for Eight Compute Nodes Reading ~30MB**

Figure 7.13: PFSread Trace for Eight Compute Nodes Reading ~40MB

Follow-Up

Our analyst had some reservation about the change in read time above 30MB. This boundary included a system reboot, and there is the possibility that the change is related to an altered system state, rather the read size per se. A follow-up test indicates that this may be the case.

52

Two additional runs were made with eight compute nodes sorting ~40MB, and ~30MB (files sesame_19990430181817 and sesame_199990430182938). The application-level read rate for the 30MB case was 2.66MB/s, in good agreement with the previous result. But the rate for 40MB was only 1.66MB/s, much slower. The write rates were 4.05MB/s and 2.61MB/s respectively. The average time between read completions for the 30MB case was 0.69 seconds on Node 1, and 0.52 on Node 2. Writes were 0.45 and 0.38. For 40MB the times were 0.90 and 0.92 for read, 0.50 and 0.64 for write.

7.4.4 Is There a Problem?

The results so far show a definite problem with low performance on Node 1, at least under certain conditions. It may be that this is related to the Size/Node-Number combination at which the problem resolved, but it might also be due to external factors. The system was rebooted at the 30/33MB boundary, and it is possible that some unrecognized mounting problem was resolved.

Aside from this, however, the results from these tests show the system functioning nearly at the speed of the disk drives themselves. The system, overall, seems to be performing fairly well.

But other tests have shown that PFS performance is highly contingent, and subject to rather complex variation. Record size has been found to be critically important.

Performance Regions

Figure 7.14 shows the results of an early test. The file system benchmark iow was used to measure the time to write 64MB of data to the PFS from one compute node, with different record sizes and numbers of disks. Record sizes were the powers of two from 64 bytes to 64 megabytes. The figure shows the results graphed as effective throughput by record size on a log-log scale.

No less than five performance regions can be identified:

- Up to a record size of 1KB the performance is abysmal. The mean time to write a record is fixed at ~19msec; the number of I/O nodes (disk stripes) does not matter. For the smallest record size of 64 bytes, it took 5.6 *hours* to write the file; for comparison, a standard SPARCstation 5 required less than 3.1 *minutes*.

- At 2KB and 4KB, the time to write a record drops to ~11msec, but there is still no appreciable difference with number of disks.

- From 8KB to 32KB, performance rises, and the different numbers of disks show different throughputs. But scaling is not even, 3 disks perform better than 1, 2 perform better still.

- From 64KB to 1MB the system appears to have achieved optimum performance. Throughput is flat, and scaling is perfect, at ~2MB/s per disk.

- Above this performance begins to drop. By 4MB with 3 I/O nodes, and 8MB for 2 and 1, performance decreases notably. By 64MB it appears to be converging for all numbers of I/O node, but this is well beyond the memory limit of the compute1 node.

53

Cross-file-system 1 node record size test 1; Parts 2-4.
File: sesame_19980406132351.iow sesame_19980407093925.iow sesame_19980408095010.iow

Target System: Paragon <sesame>   Instrumentation: iow
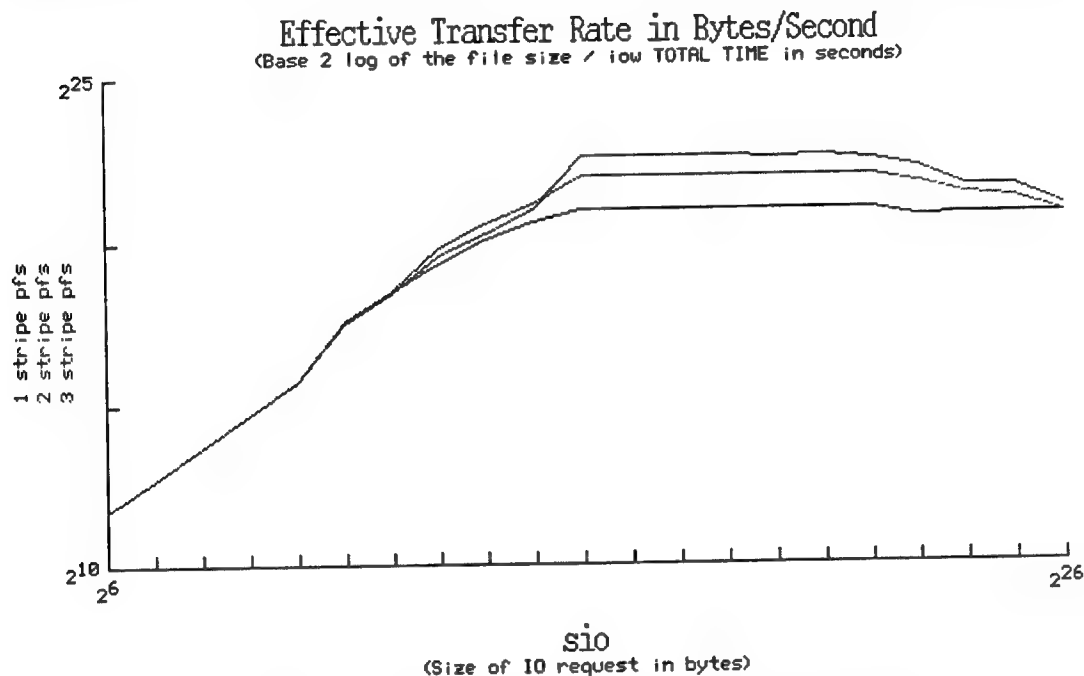Begin Date: Mon Apr 06 13:23:52 PDT 1998   Contact: Mark D. Strohm <strohm@cs.ucla.edu>

Effective Transfer Rate in Bytes/Second
(Base 2 log of the file size / iow TOTAL TIME in seconds)

sio
(Size of IO request in bytes)

Figure 7.14: PFS Write Transfer Rate with One Compute Node

Inter-Record Variation

A major feature of the performance graph is the abrupt "elbow" at a record size of 64KB. Analyses by the Modeling team predicted a more gradual roll off, with greater dependence on the number of disks (peak performance on systems with larger numbers of disks was expected to be reached at larger record sizes). To follow up on this, the Repository staff ran a more detailed series of tests on this region, looking at more closely spaced record sizes, and measuring the times for each record. Sixty-four record sizes were tested, multiples of 4KB from 4KB to 256KB.

The results were surprising. For record sizes in even multiples of 64KB, the first records were written very quickly; after about 3MB the time for each record rose to a higher stable value. Off 64KB records showed the first records taking an unusually long time, followed by violent cycling.

Figure 7.15 shows the quick-look graph of times to write 128KB records to a single disk PFS. The vertical scale is one second. Note the early phase, with a single slow record, followed by a stable section, punctuated by two "glitches," where the cycle repeats.

54

Figure 7.16 shows the times to write 132KB records on the same scale. The initial records take longer to write, and the rest of the records flow in cycles of 16 (the number needed to write a multiple of 64KB).

PFS "Elbow" test 1; part 4 of 4.
File: sesame_19980821155550.iow1

Target System: sesame <paragon>   Instrumentation: iow1
Begin Date: Fri Aug 21 15:55:51 PDT 1998   Contact: Mark D. Strohm <strohm@cs.ucla.edu>

Quick-look: time-iow1, mt2/Record Number
(iow1 benchmark invoked from time(x) — Timing of each i/o operation (microseconds))

File 0, creat on /pfs/iow/ : 1 <MAXTOR MXT-1240S   I1.2>
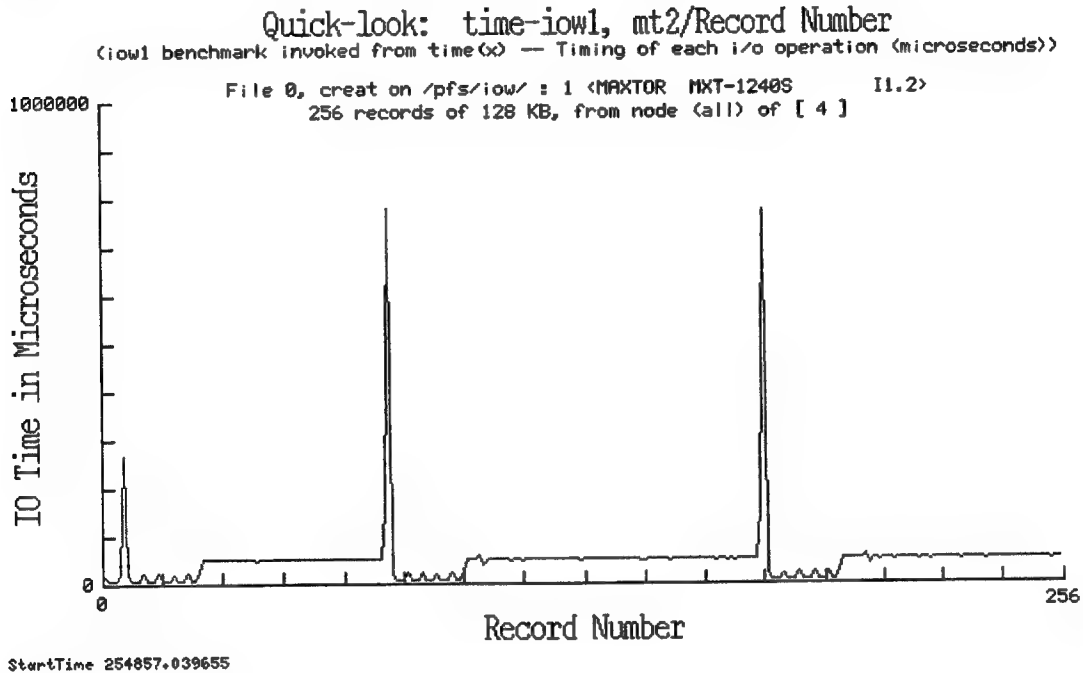256 records of 128 KB, from node <all> of [4]

StartTime 254857.039655

Figure 7.15: Write Times of Individual 128KB Records on a One Node PFS

55

PFS "Elbow" test 1; part 4 of 4.
File: sesame_19980821155550.iow1

Instrumentation: iow1

Target System: sesame <paragon>
Begin Date: Fri Aug 21 15:55:51 PDT 1998

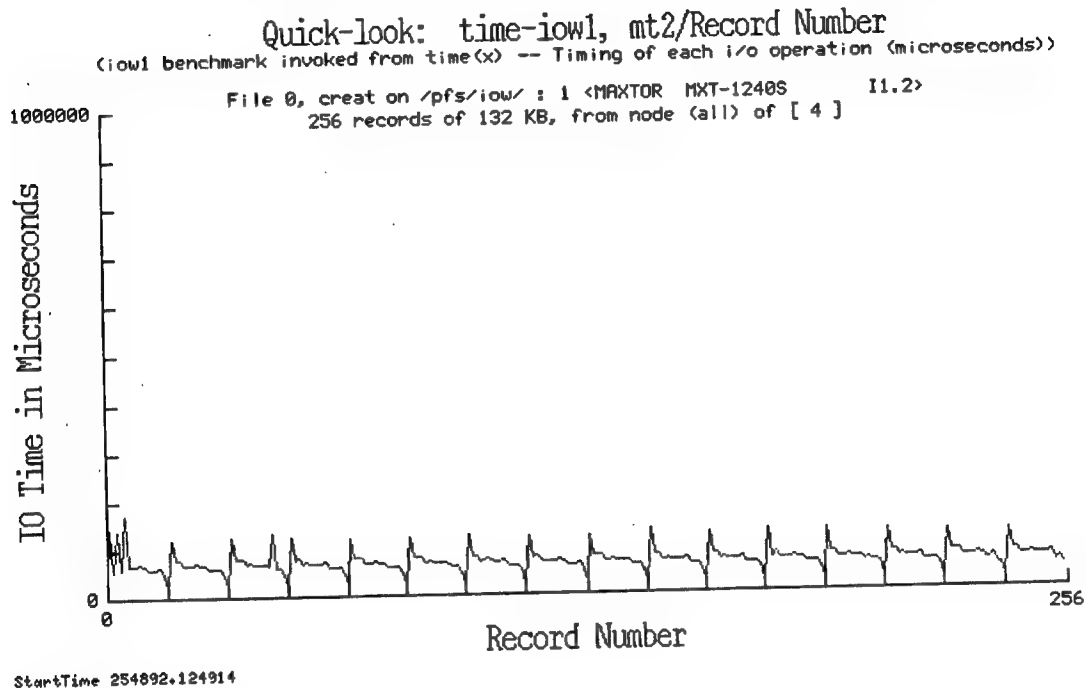Contact: Mark D. Strohm <strohm@cs.ucla.edu>

Figure 7.16: Write Times of Individual 132KB Records on a One-Node PFS

An Undocumented Buffer?

The remarkable thing about the early phase of the 128KB test is the extreme speed at which the records were written. The Paragon PFS system is documented as being unbuffered (there are special commands required to activate a buffer). This would imply a tremendous initial disk transfer rate, which for some reason, fails after a few Megabytes.

A further test with the SESAME hooks in place reveals the full extremity. Table 7-6 shows the initial phase of writing 128KB records to one PFS disk (files sesame_19981005160717.iow1 and sesame_19981005165206). From the application level we see times below 7 milliseconds. From the system level, we see values as low as 1 msec.

56

Table 7-6: PFS Write Times in Milliseconds for Initial 128KB Records

Record #	iow1	SESAME
0	16.983	1.126
1	16.098	1.283
2	6.895	1.277
3	6.803	1.002
4	6.620	1.005
5	22.130	1.037
6	268.203	262.197
7	16.799	1.127
8	6.954	1.095
9	7.017	1.121
10	6.971	1.112
11	21.837	1.135
12	7.312	1.120
13	7.692	1.111
14	7.259	1.113
15	21.524	1.394
16	7.027	1.336
17	6.747	1.126
18	6.768	1.132
19	19.366	1.377
20	7.015	1.374
21	7.005	1.382
22	7.025	1.374
23	21.428	1.142
24	6.994	1.135
25	7.004	1.127
26	25.117	19.181
27	48.079	42.116
28	48.034	41.928
29	48.143	42.177
30	50.791	45.272
31	48.266	41.984

The SESAME values demonstrate that, counter to our clear understanding, the data is being buffered. Writing 128KB directly to the disk in 1 millisecond simply is not possible for the hardware. We can even isolate the location of the buffer: it is not in the OS proper, because the SESAME numbers show it in the write completing, and it is not on the disk, because the data could not have crossed the SCSI bus. Therefore it must be located in or near the device driver.

This situation presents a veritable catalog of potential interpretation errors. If we write 3MB (records 0 to 23), which seems a substantial file, the timings would suggest a throughput of 5.6MB/s—about the maximum for the disk. Discarding record 6 as an obvious "outlier" would bring the estimate to over 10MB/s, the speed of the SCSI bus. Looking for a peak value would give over 20MB/s—suspiciously high, but a *measured number* none-the-less.

Only the SESAME numbers, with throughputs up to 120MB/s, force us to reevaluate our understanding. Forty-five milliseconds per record gives a throughput of 2.78MB/s, a very reasonable value for the disk hardware used.

With just the application-level values, we would be trying to explain the mysterious drop in performance after the initial phase. With the SESAME data we are searching for an undocumented feature. SESAME gives us a complete change in perspective.

7.5 Summary Measurements: VM and CPU

The analyses done above have relied on "trace" records, that is, precise data on each instance of an event. Tracing requires transporting a significant amount of information for each instance. For relatively rare events, the amount of data remains small and can be stored and transported with minimal effect on the system. Common events, however, are too numerous to use this technique without noticeably impacting system performance.

Two classes of such common events were measured in the sesame-sort tests: Virtual memory activity, and CPU loading.

7.5.1 Virtual Memory Activity, the Major Performance Barrier

A significant feature of the sesame-sort data is a general performance roll-off when the sort size reaches approximately 8MB per compute node, followed by a very sharp drop in performance at about 10MB/node. Since each compute node has only 16MB of RAM, and must hold the OS, the program, and the data, it is likely that these performance drops are related to RAM exhaustion and the attendant rise in swapping. This effect should be easily detectable with the SESAME VM instrumentation.

The VM instrumentation was used to produce "summary" records. Eighteen different types of VM Page Faults are recorded. For each type, the summary record shows the number and average duration of the events that ended during the pull interval. Records were pulled from each node approximately every five seconds.

A global measure of VM activity, "Virtual Memory Faults," was computed by summing the event counts for three of the measured numbers: "ExecPageFaults," the total number of faults associated with the program's executable code; "FilePageFaults," faults associated with file operations; and "IntPageFaults," faults associated with internal OS activity. For each pull interval this should yield the total number of VM Page Faults. On each compute node this number was found for all pulls that included program operation, beginning with the first pull after sesame-sort issued the "start" stamp, and ending with the pull after the "end" stamp. And all these were summed to produce a global measure of the total VM activity during the program's run.

Figure 7.17 shows this measure displayed by number of compute nodes, and size of sort per node. The measure rises with both number of nodes, and size of sort. For the one- and two-node cases, the value rises very sharply at 10MB/node. The lower sizes show a smoother curve. The curves for larger numbers of nodes are notably less stable than for small numbers.

Because the sort size is defined per node, it is reasonable that larger numbers of nodes should produce larger amounts of activity, though no precise scaling expectation has been calculated. Similarly, larger numbers of nodes produce a less stable pattern of program execution (the amount of activity required depends on the precise timing relationships between the nodes), so a total VM measure would be expected to show rougher curves.
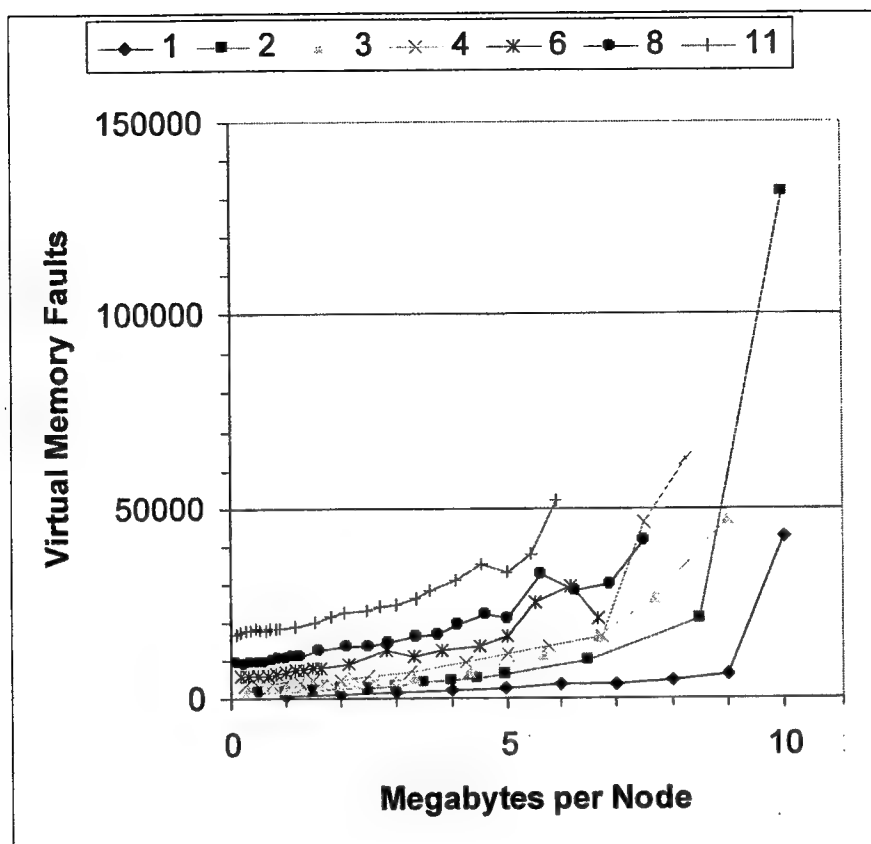
Figure 7.17: Total Number of Page Faults by Sort Size

Main Performance Curve, No 8MB Limit

For most of the size range observed, the curves show a smoothly increasing rate, following roughly an exponential growth pattern. Figure 7.18 shows the data for the three-compute node case, with an exponential trend line fitted. The fit is very good; size per node explains over 99% of the variance in Page Fault count (in the graph, the data points are difficult to even see behind the trend line).

The lines show no sharp curve at the 8MB/node level. This suggests that the associated performance roll-off is part of the underlying trend, rather than a fundamental change in behavior.
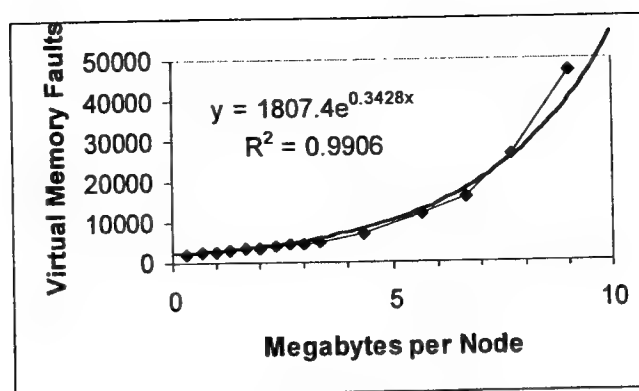
$$y = 1807.4e^{0.3428x}$$

$$R^2 = 0.9906$$

Figure 7.18 : Total Page Faults by Sort Size, for Three Compute Nodes

The 10MB/Node Boundary

The 10MB/node boundary, however, shows a clear break from the main performance curve. The boundary is particularly clear for the single compute node runs, with a smooth, slowly rising curve, which abruptly jumps to several times its previous level.

T-7 and Table 7-8 show the breakdown by event type for the 9 to 10MB/node boundary for this case (files sesame_199901201113229 and sesame_199901201115010). Most categories either have no instances of the event type, or show only modest increases. The large increases occur in the swapping related faults: Internal Page-Ins, Page-Outs, and Reactivations. This is consistent with RAM exhaustion.

Table 7-7: VM Event Counts by Type, One Compute Node, ~9MB

Type	Page-In	Page-Out	Copy-on-Write	Reactivation	Zero-Fill
Executable	0	0	0	28	0
File	195	0	0	0	1
Internal	1458	1946	1186	4002	182

Table 7-8: VM Event Counts by Type, One Compute Node, ~10MB

Type	Page-In	Page-Out	Copy-on-Write	Reactivation	Zero-Fill
Executable	0	0	0	96	0
File	216	0	0	0	1
Internal	20171	19490	1314	50551	181

Unfortunately, the numbers also reveal a problem with the data. Comparing the values with the corresponding data points in Figure 7.17, we can see that the total number of events in the tables is higher than the supposed grand total. The total number of faults shown for the 10MB case in

the figure is less than 50,000, while the number of internal reactivations shown in the table is over 50,000. This should not be the case.

Clearly, our understanding of the VM code is incomplete. In a continuing project this would be a valuable finding, indicating a potential trouble spot with the OS itself (or a bug in out instrumentation). Regrettably, time does not permit investigation.

7.5.2 CPU Loading, Unexpected Library Behavior

The other summary measure used with sesame-sort was CPU loading. This measurement was unusual in that it did not involve the placing of "hooks" in the OS, but rather the collection of the native CPU data from the OS. The Paragon samples the CPU state 100 times per second, and notes it as serving a user process, a system process, or being idle. The SESAME subsystem averages these different activities over the pull interval, and reports them in a summary record. This allows us to map the CPU use over the execution of the user program, though at a rather low time resolution.

Figure 7.19 shows a special plot developed for sesame-sort data, which combines the application data, with composite measures of the CPU and VM data. The data is from the middle node of three, sorting 13MB (file sesame_19990121104605.sort-merge).

The vertical color bands show the sesame-sort activity. The pale green band represents the node reading the input data. The pale yellow bands show when the node is sorting its numbers. The bands in between the sorts show the node trading sorted numbers with the node below ("ask" events), and with the node above ("answer" events). The pale blue band at the right indicates the node writing its fully sorted numbers to the output file.

The red line (running toward the top of the graph) shows the average CPU loading for the pull interval.

The blue line (running along the bottom of the graph) shows a composite measure of virtual memory activity. This measure was intended to estimate the proportion of the pull interval occupied by VM faults. It was computed by taking the events described (the three categories of page faults), multiplying each by its average time, summing to get the total time, and dividing this by the pull interval.

In this case, the plot shows a very satisfying pattern. CPU loading is nil while the application is not running; moderate during I/O, and very high during the pure-computation of the sort phase.
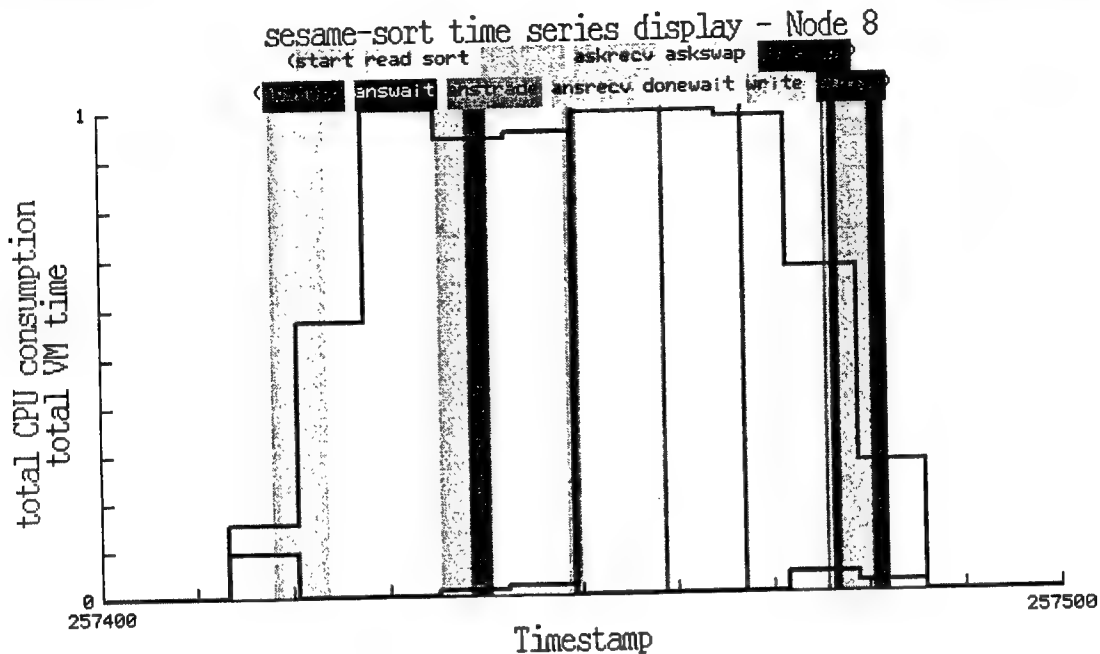
Figure 7.19: Typical CPU and VM by Sort-Phase Plot

Figure 7.20 shows the plot for a more problematic case. This data is from the top node of 11, sorting ~50MB (file sesame_19990129130409.sort-merge).

As in the first example, we can see the read and write at the ends, and alternating sorts and trades in the left section of the plot. The right section is dominated by a long period of "donewait." In this phase, the node has completed sorting its numbers and has found no more to trade. It is waiting for an "exit" signal from the lowest ranked node before proceeding to open the output file (which requires a global synchronization).

There are two disturbing features of this plot: the two "out of range" spikes in the VM line, and the 100% CPU loading during the wait.

In two highly visible instances, the VM line extends above the normal plot area (through the titles, and beyond the edge of the image). The measure (the total amount of time spent in VM activity, divided by the total amount of time) should represent the proportion of the pull interval spent in VM activity. But the computed value was often above unity; more time appears to have been spent in VM activity than was available.

One reason for this was known during subsystem design. Since the record is based on events that ended in a pull interval, unusually long events could cross boundaries. An event lasting ten seconds, being reported in a five-second pull, would give the impression of over unity time consumption. Another potential source of this is multithreading. If several VM events are

62

pending simultaneously, then they will be occupying more "total" time than elapsed time. But the data may also suffer from the same issue observed above. We may be overestimating the number of events, and hence the amount of time consumed.

The period of very high CPU consumption at the left was also a surprise. At this time the user program is in a wait-state, using the MPI_Probe function. This function blocks until a message is available. It was assumed that this would put the application into an interrupt wait, with low CPU consumption. Instead we see a maximal consumption phase, suggesting that the MPI function is testing for incoming messages with continuous polling.

The implication of this for performance measurement lies in the ability to distinguish useful activity. We had been planning to use CPU loading as a figure of merit for effective use of the compute nodes. The ultimate goal of the system is the *computation* performed, and the final limiting factor on this is the CPU speed. The goal of the OS, and the system as a whole, is then to keep the CPUs fully loaded, preferably on the user process itself.

At the user level, we used the "time" utility to record the total amount of time the compute nodes spent in user and system operation, and the elapsed time for the run. Our analyst had hoped to use the proportion of total elapsed time that the processors were active as a figure-of-merit for the program, and apply the SESAME data to separate the slack time caused by inefficiencies in the algorithm, from that caused by inefficiencies in the system.

In this conception, the long wait period seen in the graph is a severe inefficiency caused by the algorithm. But the high CPU loading revealed by SESAME shows that values do not track useful activity. The analyst's "reasonable guesswork" was wrong, and the intended user-level measure worse than useless. With only user-level measures, this problem would be subtle; with the SESAME data, it is unmistakable.
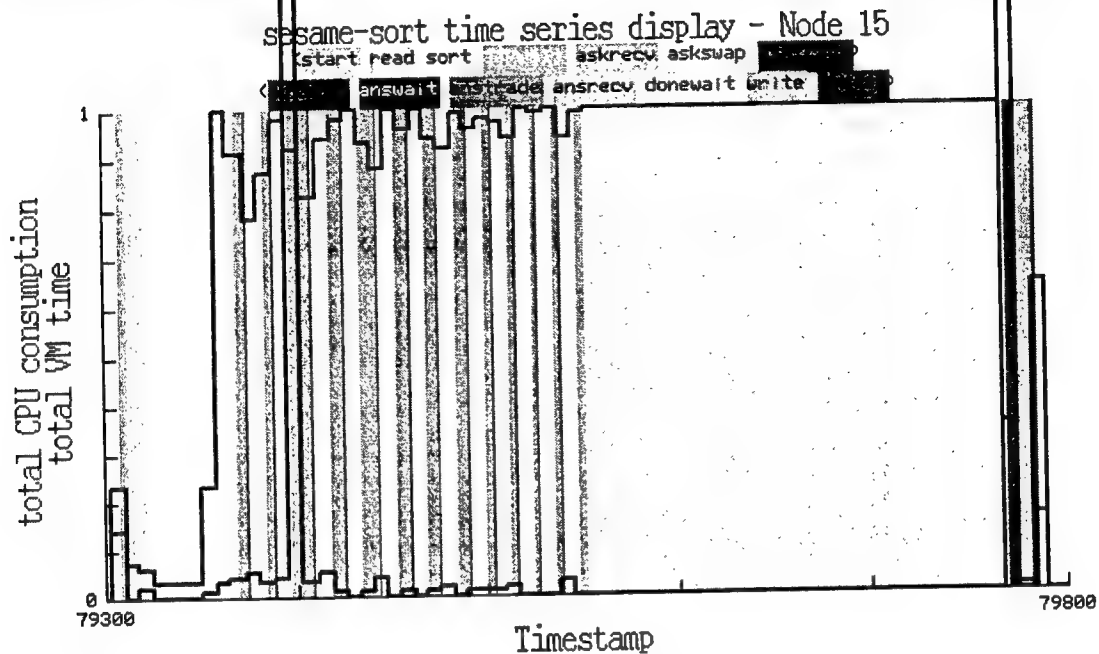
63

sesame-sort time series display - Node 15

Figure 7.20: Sort Plot with VM Spikes and High CPU Loading During Wait

64

8. MODELING RESULTS

8.1 MPI-SIM

8.1.1 Benchmarks

We have validated MPI-SIM and measured its performance for the NAS (Numerical Aerodynamic Simulation) Parallel Benchmarks (NPB 2) (Bailey et al. 1995), a set of programs designed at the NASA NAS program to evaluate supercomputers. The IBM SP2 at UCLA was selected as both the target and host machines. Each node of the IBM SP2 is a POWER2 node with 128Kb of cache and 256Mb of main memory. Nodes are connected using a high-performance switch, which offers a point-to-point bandwidth of 40Mb/s and has a hardware latency of 500ns. The NPB benchmarks are written in Fortran 77 with embedded MPI calls for communication. Since MPI-SIM currently supports privatization only for C programs, it was necessary to convert the benchmarks to C. We were able to convert four out of the five benchmarks using f2c (Feldman et al. 1990), a Fortran-to-C converter. The specific configurations of the benchmarks that were used in the performance study were constrained primarily by their memory and CPU requirements. Table 8-1 summarizes the relevant configuration information for the benchmarks. Each benchmark was executed for three target machine configurations. For example, LU was executed on 4, 8 and 16 processors.

Names	Lines	Class	Target Procs..	Target 1	Target 2	Target 3
				Target Program Size/Simulator Size (Host Procs. for Simulator)		
LU	4623		4,8,16	14M/57M (1,2,4)	8M/32M(2,4,8)	5M/18M(4,8,16)
MG	2712	S	4,8,16	600K/8M (1,2,4)	400K/5M (1,2,4,8)	300K/3M (1,2,4,8,16)
BT	6290	S	4,9,16	2M/24M (1,2,4)	1M/15M (1,2,4,9)	1M/12M (1,2,4,9,16)
SP	5555	S	4,9,16	700K/7M (1,2,4)	500K/6M (1,2,4,9)	500K/5M (1,2,4,9,16)

Table 8-1: NAS Benchmarks

8.1.2 Verification and Validation

The target programs and the simulators were executed for all processor configurations listed in Table 8-1. For each target and host processor configuration, each simulator was executed in four modes described in Section 8.1.3. The NPB 2 benchmarks are self-verifying, meaning that each benchmark after completion compares the computed results against precomputed results to ensure that it executed correctly. All target programs and simulators were found to verify correctly.

Figure 8.1 plots the target program execution time (solid line) and the execution time as predicted by the simulator (dashed lines) as a function of various target machine configurations. Note that the simulator predicted times are plotted for each host configuration listed in Table 8-1. The graphs were nearly identical in all simulator modes, and consequently the figure shows only one mode: the NMP+CEP+Det mode. In the best case, the predicted and measured times differed by less than 5% and in the worst by 20%, thus lending reasonable credibility to the simulations.

65

8.1.3 Simulator Modes

A simulator can be executed in four modes. In three of these the simulation status is non-deterministic, differing in the use of the protocol for EIT advancement: the CEP mode (uses conditional event protocol), NMP mode (uses null message protocol), and CEP+NMP mode (combines both). In the last mode the simulation status is deterministic and both the conditional event and the null message protocol (CEP+NMP+DET mode) are used. These simulator modes allow us to determine the contribution of each protocol and each optimization to the performance of the simulation.

8.1.4 Reducing Synchronizations

We compared all modes of each simulator against the traditional quantum protocol. Performance of the simulation protocol in each simulator mode is gauged by the number of rounds of protocol messages, R, sent per processor. The performance of the quantum protocol is measured as the number of global synchronizations it takes to simulate the same target program. A round of protocol messages is similar to a global synchronization, although it is frequently less expensive, since in many cases a processor does not need to wait to receive protocol messages from all other processors.

Given a target processor configuration, we found that R decreases only modestly as the number of host processors used to simulate the configuration is increased. Figures 8.2, 8.3, 8.4, and 8.5 show the variation of R with the simulator modes for two representative target and host processor configurations of each benchmark. In each graph, the number of rounds of protocol messages is normalized against the number of global synchronizations of the quantum protocol. The X-axis
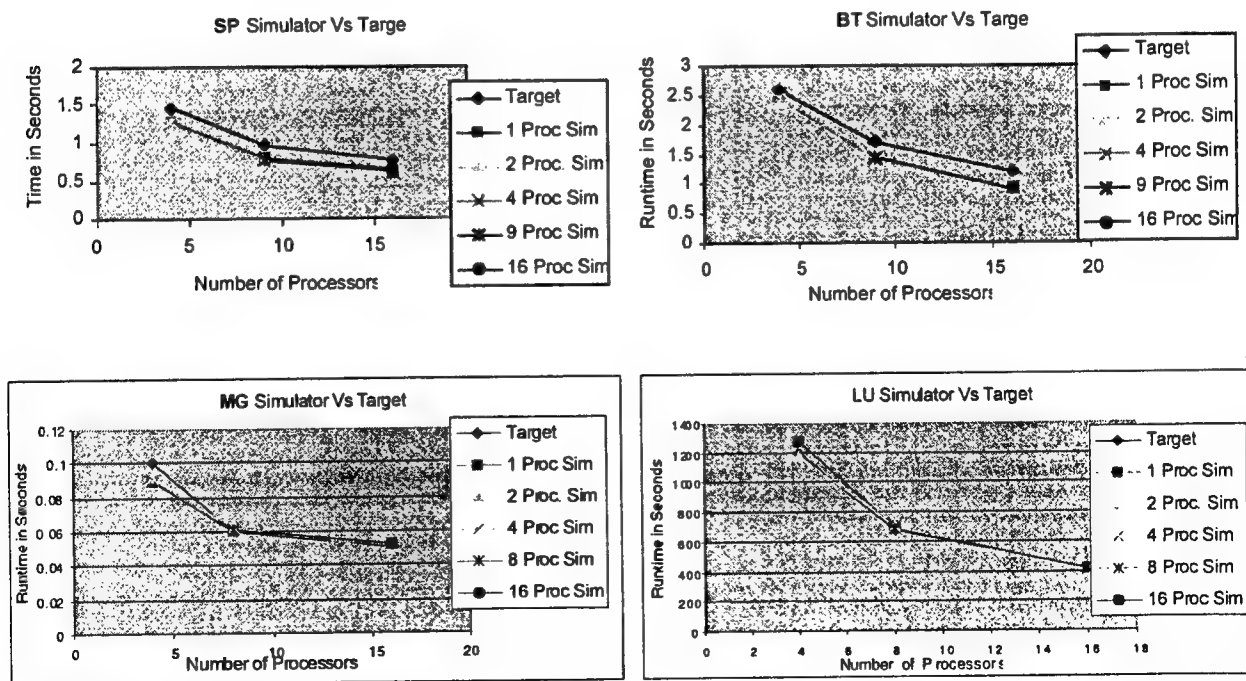

Figure 8.1: Target Execution Time vs. Simulator Predictions for NAS Benchmarks

66

shows the simulator mode, where "N+C" refers to the NMP+CEP mode, and the "N+C+D" mode refers to the NMP+CEP+Det mode.
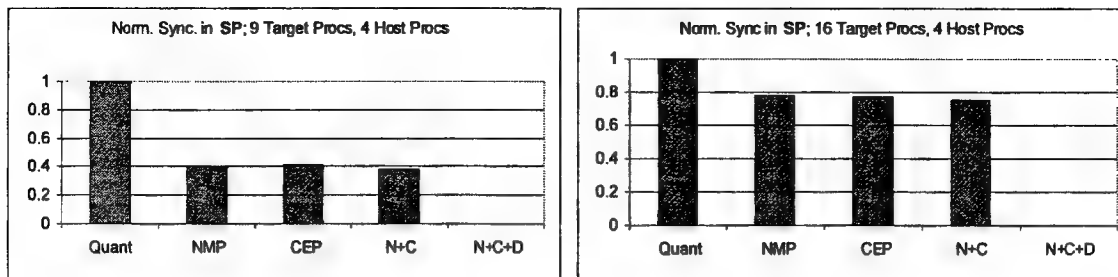

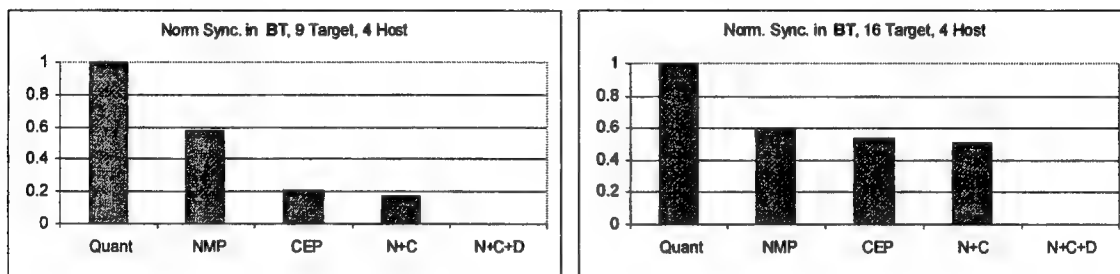
Figure 8.2: Performance of Simulators for SP

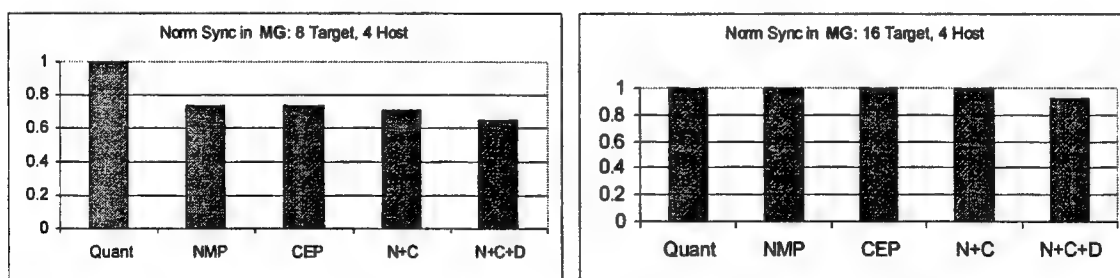**Figure 8.3: Perfomance for Simulators for BT**

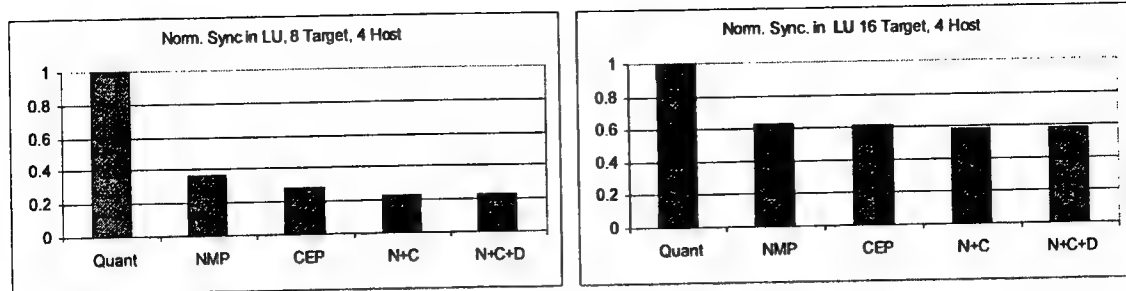**Figure 8.4: Performance of Simulators for MG**

67

Figure 8.5: Performance of Simulators for LU

Consider only the CEP mode: the amount of improvement over the quantum protocol is strongly dependent on the average duration for which an LP (i.e., thread) executes before getting blocked. Table 8-2 shows this average duration for each benchmark and each target program configuration. L is the minimum message latency of the target machine. The 9-processor BT benchmark has the largest average uninterrupted execution time per thread, and in the simulation the CEP mode is
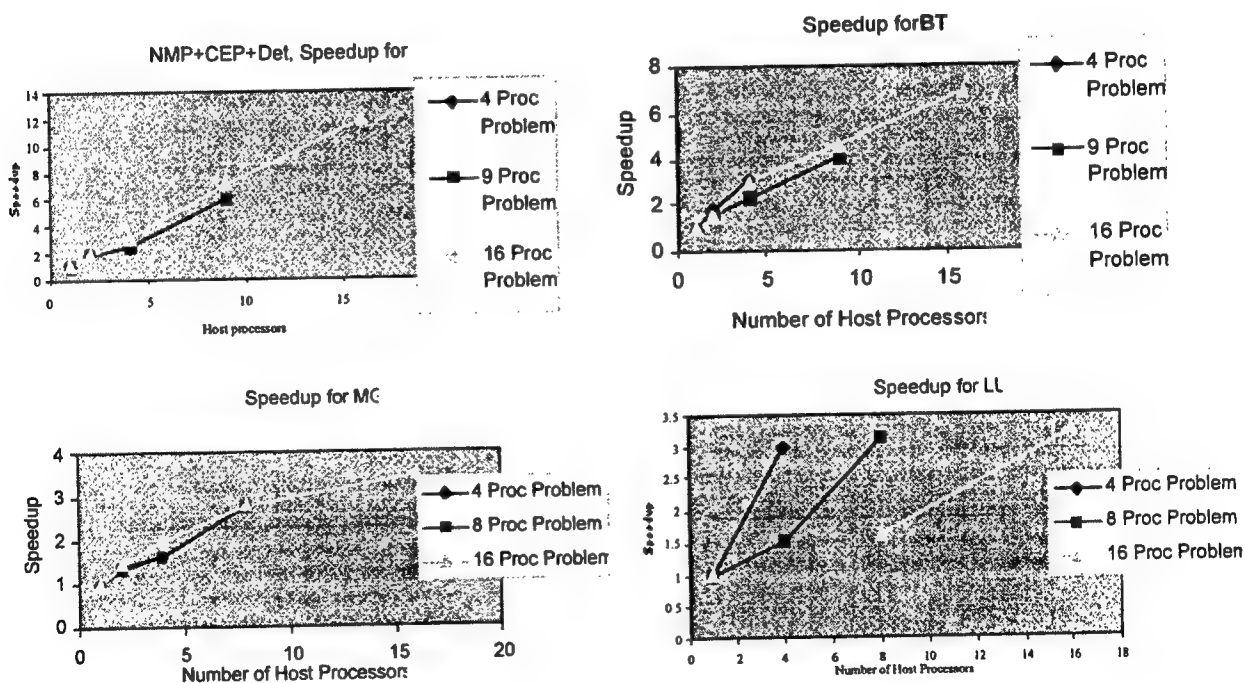

Figure 8.6: Fast Simulator Speedups

able to eliminate more than 80% of the global synchronizations of the quantum protocol. The NMP mode is able to eliminate only 40% of the global synchronizations of the quantum protocol. This is because the CEP significantly improves over the NMP when some LPs are far ahead of

the others in simulation time, requiring the other LPs to exchange many rounds of null messages to update their simulation times. The 16-processor MG benchmark has the smallest average uninterrupted execution time per thread, and the NMP+CEP mode is unable to significantly reduce the number of global synchronizations of the quantum protocol.

Benchmark	16 Target Procs.	8 or 9 Target Procs.
LU	8.74L	11.77L
MG	2.79L	4.03L
BT	12.33L	24.81L
SP	4.61L	9.29L

Table 8-2: Average Uninterrupted Execution Time

Using our optimizations for exploiting the determinism in the program, we note that it is possible to eliminate *all global synchronizations* in the BT and SP benchmarks. The optimizations were not effective in significantly reducing the synchronizations from the MG and LU benchmarks as discussed in the next section.

8.1.5 Reducing Simulator Execution Times

We present the speedup measured by executing the parallel simulator using the combined NMP and CEP algorithm as well as the deterministic protocol. A receive can be deterministic if it either specifies the source explicitly or it specifies an explicit tag and each source uses unique tags. Although the first type of determinism can be detected automatically by the current simulator, we have not yet implemented the second mode. Out of the four benchmarks used, SP and BT have the determinism of first type. The MG and LU benchmarks have determinism of the second kind. Although this optimization is not automatically implemented in the compiler, we manually inserted the optimizations to evaluate the potential benefit that can be derived from exploiting this form of non-determinism. The final speedups obtained from the execution of all the benchmarks are presented in Figure 8.6. We measure speedup (N) by taking the ratio of the execution time of the sequential simulator to the execution time of the simulator using N processors. The speedups for the LU benchmarks are relative to the smallest host processor configuration that could be used to run the simulator. For example, the 8-target-processor simulator could be executed on 2, 4 or 8 host processors. Hence, the reference execution time is of the 2-processor simulation. This *understates* the expected performance improvement for this application. Notice that the speedups achieved with the simulation are characteristic of the application itself, as the simulation overhead is relatively small.

8.2   Parallel I/O

8.2.1   Benchmarks

The workload for our experiments was provided by using three MPI/MPI-IO programs plus a synthetic benchmark. The first two programs are from the NAS BTIO Parallel I/O Benchmarks v0.1. 3 The BTIO benchmark extends the original BT benchmark (Bailey et al. 1995) by using MPI-IO to write data to a file at regular time intervals. The first BTIO program is the "simple" benchmark, where only non-collective I/O and primitive MPI datatypes are used. The second is the "full" benchmark, where data is fully described using MPI non-contiguous, user-defined datatypes, and collective I/O is used. Both benchmarks exhibit numerous seek operations. The third benchmark we have used is a basic out-of-core matrix multiplication application. Lastly, the

synthetic benchmark generates random read/write requests with the starting address, request size, and read/write ratio generated by stochastic functions.

8.2.2 Speedup with Parallel Execution

Figure 8.7 shows the simulator speedup from parallel simulation of the NAS BTIO Simple and Full Benchmarks using the simple disk models. For the full benchmark, all three implementations of collective I/O are shown: Global Barrier (GB), Node Grouping (NG), and Two-Phase I/O (2P). Note that these results only address the issue of simulation performance and of the underlying collective I/O implementation.

Each graph shows the NAS Benchmark with three different configuration sizes: 4-processor problem, 9-processor problem, and 16-processor problem. Each configuration is run through a simulation with a varying number of processors. Note that there is considerably more speedup in the NAS Full Benchmark when using Two-Phase I/O than there was when using Global Barrier and Node Grouping. This occurred because the computation granularity is much greater in Two-Phase I/O, predominantly from the work required in the permutation phase, where the numerous small requests are merged into fewer, larger requests.
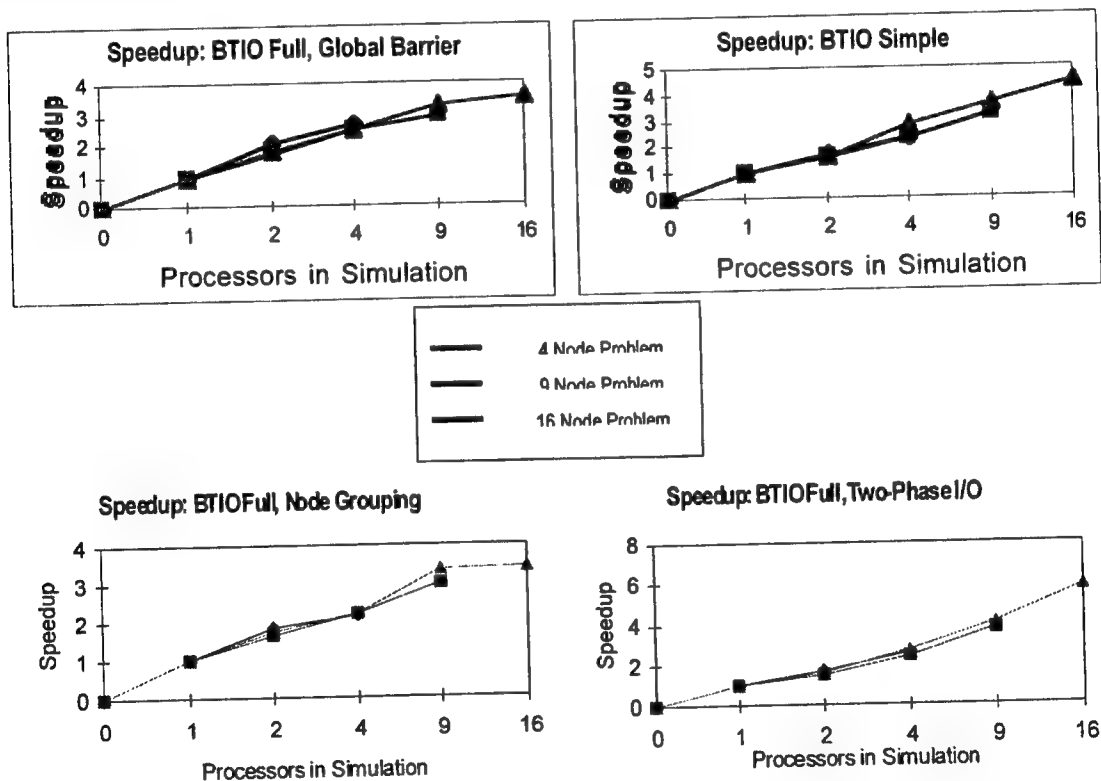

Figure 8.7: Simulator Speedups for NAS BTIO Benchmarks

8.2.3 Effect of Collective I/O

Also using a simple disk model, Figure 8.8 shows a comparison of the execution time as predicted by the simulator for different collective I/O implementations. Although node grouping has previously been shown to yield up to eight times improvement (Nitzberg 1992), it has

performed very poorly for the NAS Benchmark. This is mostly due to the fact that this particular problem size of the NAS Benchmark does not generate enough I/O requests to flood the interconnection network or cause any resource contention. Thus, by not allowing all the LPs to make their I/O requests simultaneously as in global barrier collective I/O, we are simply delaying the execution.
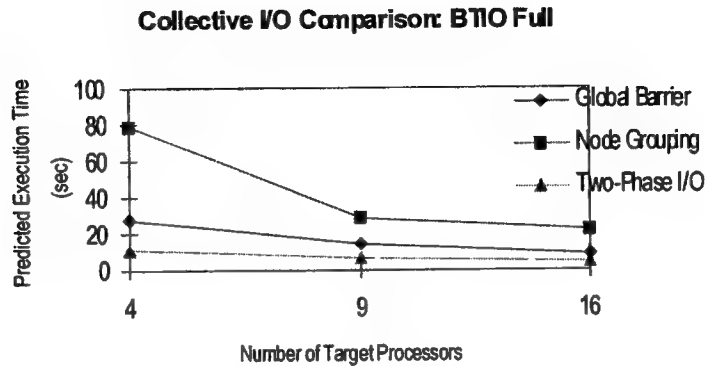

Figure 8.8: Effect of Collective I/O

The predicted time for two-phase I/O is clearly the best among the three implementations. This can be attributed to its ability to take the non-contiguous datatypes used by the BTIO Full Benchmark and merge them into larger, contiguous requests. Table 8-3 depicts the actual number of total seek and write requests made by the benchmark using global barrier and two-phase I/O. Node grouping is not listed in the table, since it generates the same number of requests as global barrier.

NAS BTIO Full Benchmarks						
Target Procs	Global Barrier Totals			Two-Phase I/O Totals		
	Seeks	Writes	Avg Req Size	Seeks	Writes	Avg Req Size
4	20976	20736	40 bytes	48	48	17280 bytes
9	21492	20736	40 bytes	108	108	7680 bytes
16	22464	20736	40 bytes	684	684	1213 bytes

Table 8-3: Simulator I/O Statistics: Global Barrier vs. Two-Phase I/O

The main components of disk access time are the time required to move the disk head to the correct track (seek time), the time spent waiting for the correct disk sectors to rotate beneath the head (rotational latency), and the time needed to move the data to/from the disk (data transfer time). Seek time and rotational latency are overheads which must be paid for each disk access and may constitute a significant portion of the I/O service time, particularly for small requests. By merging many small requests into a few large requests, overheads are reduced significantly. Also, the minimum amount of data which may be transferred to/from a disk in a single operation is a disk sector (typically 512 bytes). Requests which are smaller than this will result in more data being read than is necessary. However, when a number of these small requests access the same sector, they may be grouped together, reducing (or even eliminating) the amount of excess data which is read.

8.2.4 Effect of Caching Policies

For initial testing of each of the cache management policies, we used the synthetic benchmark to generate the workload. The workload consisted of 5000 warm-up requests and 5000 measured requests, with each request generally consisting of between one and four data blocks. The percentage of read requests was set to 80%.

Figure 8.9 shows the execution time of this synthetic benchmark as the number of ionodes in the system is varied from 1 to 16 (the number of cnodes is held constant, either 4 or 8):
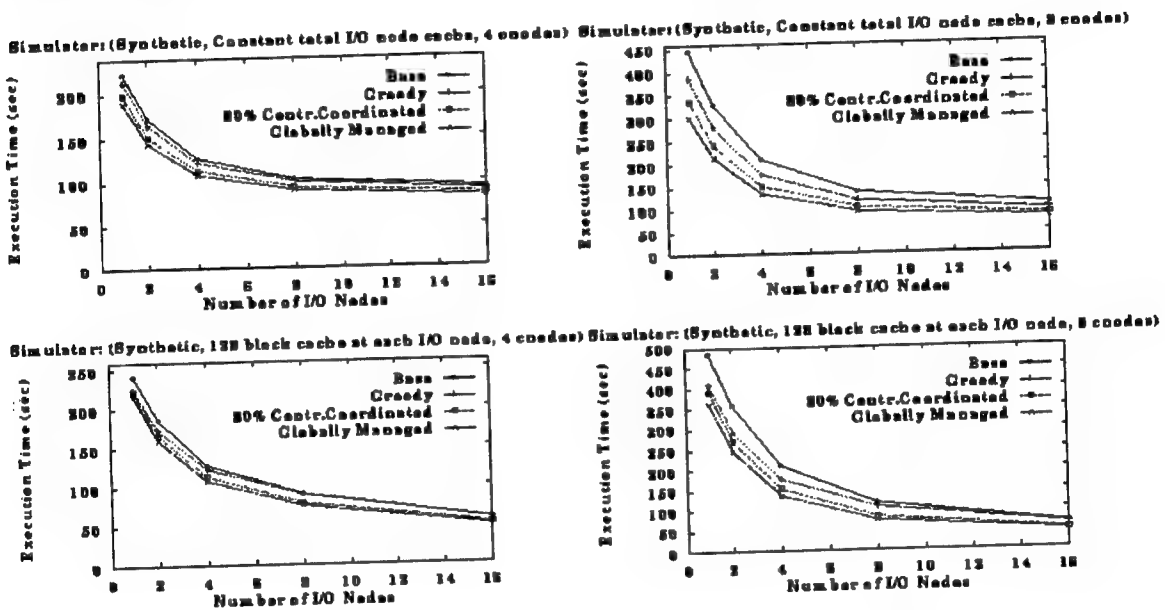

Figure 8.9: Predicted Execution Time for Constant and Variable Size Cache

Figure 8.10 shows the number of data blocks which had to be actually read from the disk for the same workload and file system configurations:
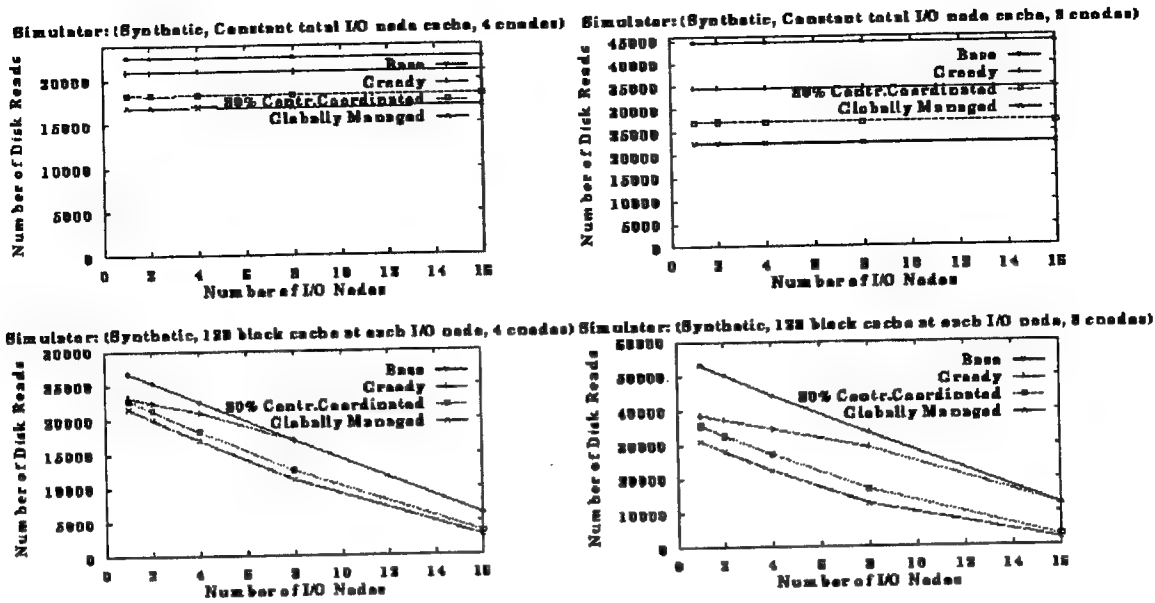
72

Figure 8.10: Number of Disk Read Ops for Constant and Variable Size Cache

Reads were chosen since this is where the major difference in performance of the various cache management policies can be seen. To eliminate any other variations due to a difference in the handling of write requests, the experiments were run using write-through and write-around, for which all cache policies behave the same. For each set of graphs, two scenarios were examined. First, the size of the cache at each ionode was fixed at 128 blocks. Second, the total cache size was fixed at 512 blocks and was divided equally among the ionodes. In this case, as the number of ionodes increased, the size of the cache at each ionode decreased. This allowed the benefit of having more disks available to be seperated from the benefit of having a larger aggregate cache size. These results clearly show that caching performance continued to improve as the level of cache cooperation was increased and refined.

Figure 8.11 shows the execution time and the number of disk reads for the matrix multiplication program, as the size of the matrices is increased, for each of the cache management policies.
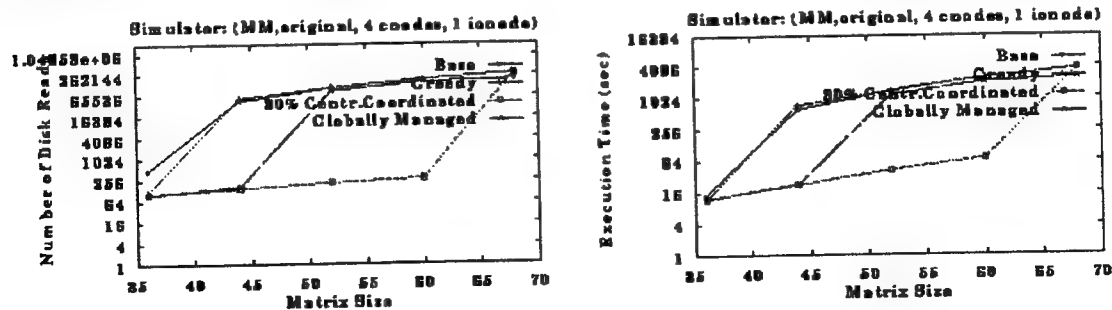

Figure 8.11: Performance of Cache Policies for Matrix Multiplication

The same file system configuration consisting of four cnodes and one ionode was used for all cases. The size of the matrices was carefully chosen to gradually become larger than the capacity

73

of the ionode cache and a single cnode cache. It was expected that this would show the fundamental difference in the way each caching scheme operates.

When the dataset fits comfortably into the ionode cache and a single cnode cache, the base caching scheme performs nearly as well as as the cooperative caching algorithms. But as the dataset becomes larger, both base caching and greedy forwarding become ineffective. This is due to the repeating sequential access pattern of the matrix multiplication program. By the time the last columns of the second matrix are read, the first columns, which must be re-read for multiplication with the next row, have been evicted from the cache. Centrally coordinated caching, however, performs extremely well. By eliminating data redundancy for a large portion of the file system's cache, all necessary data remains in the cache, resulting in compulsory misses only. Of course, once the data becomes too large to remain in the ionode cache and the centrally coordinated cache, performance becomes as poor as for the other caching schemes.

The poor performance of the globally managed caching scheme was quite unexpected, especially since it clearly outperformed all the other caching policies for the synthetic benchmark. The simulator was used to closely examine the program's actual data access pattern and the resulting cache behavior. It turned out that the program's excessive inter-process data sharing, coupled with the sequentiality of using a single ionode, works poorly with the globally managed caching's data placement policy (using the ionode cache to store evicted cnode cache blocks). Data tended to clump together in the cache of the last cnode to access the shared data, while the caches of the other cnodes remained relatively empty. Once this unfortunate cnode's cache became full, it would send evicted blocks to the ionode cache. Once the ionode cache became full, evicted blocks were discarded, even though many cnodes had space available within their caches. Solutions to this shortcoming are currently being investigated.

8.2.5   Interaction Between Caching and Collective I/O

The matrix multiplication program was chosen for the study of the interaction between the cooperative caching and the collect I/O techniques since it benefits from both approaches. In our implementation of matrix multiplication, the majority of I/O operations involve reading the columns of the second matrix. Not only does reading a single column involve numerous I/O operations (since the matrices are stored on the disks in row-major order), but every column must be read in once for each row of the first matrix. Thus, for two NxN matices, reading in each column requires N operations and all N columns must be read in N times (once for each row of the first matrix), resulting in N3 operations directed to the file containing the second matrix. The file with the first matrix is hit with only N operations since each row is read in only once and may be read with a single operation since row elements are stored contiguously on the disk (except when a row happens to cross a disk block boundary).

Two different access patterns were used for reading the columns of the second matrix. In the original pattern, each target process began reading at a different column. This staggering was done to allow some benefit from caching when all target processes read in the second matrix at the same time. The last columns needed by each process would already have been read in by other processes. The second access pattern has all target processes begin reading at column 0. This allows a collective operation to be used to eliminate redundant reads. Each column is read once and transmitted to all other target processes, though the columns are re-read when the target processes move on to the next set of rows from the first matrix.

74

The graphs presented earlier in Figure 8.11 show the execution time and number of disk reads for each cache technique using the original access pattern and no collective I/O. The graphs in Figure 8.12 show the performance with the second access pattern, though still without any collective I/O.
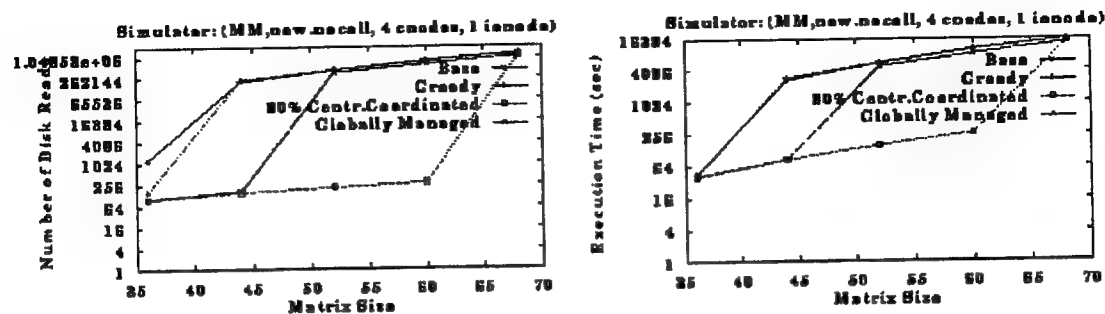
Figure 8.12: Performance of Matrix Multiplication with Second Access Pattern

As expected, cache performance was significantly worse for this access pattern since all processes needed the same data at the same time.

Next, Figure 8.13 presents results using the simple Global Barrier Collective I/O algorithm.
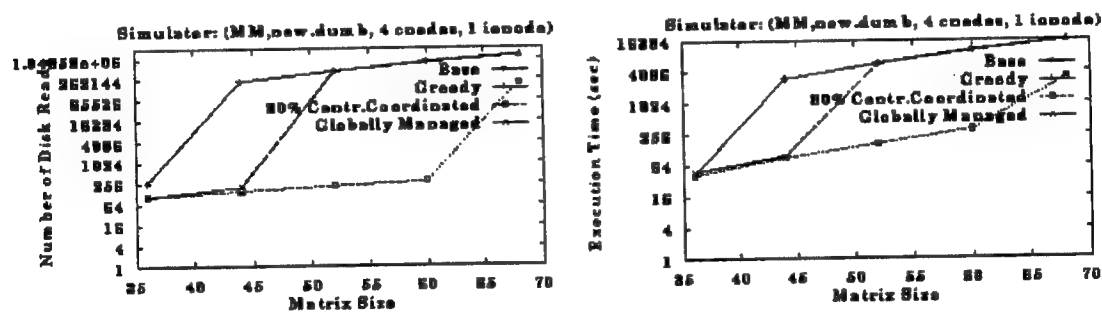
Figure 8.13: Performance of Matrix Multiplication with Global Barrier I/O

This technique does not reduce the number of I/O operations and performs much like the non-collective experiment. When the collective I/O algorithm was changed to the two-phase I/O strategy, however, significant improvements were seen, as shown in Figure 8.14.

Figure 8.14: Performance of Matrix Multiplication with Two-Phase I/O

75

By eliminating the massive amount of redundancy when reading columns of the second matrix, two-phase I/O drastically reduces both the execution time and the number of disk reads. It also appears that reducing the number of I/O operations also directl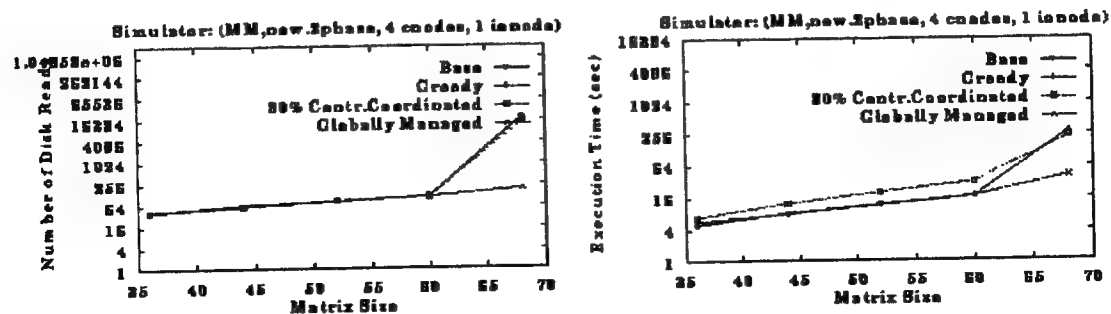y benefits the performance of caching since even the simple caching strategies performed well for all but the largest matrix size. Also, while all caching techniques required the same number of disk reads for the smaller matrices (compulsory misses), the centrally coordinated cache had a higher execution time than the other strategies. This points out the tradeoff inherent in centrally coordinated caching. While the global hit rate remained low, the local hit rate was higher due to the reduction in size of the cnode's locally managed cache. Few blocks had to be read from disk, but many had to be retrieved from remote cnode caches, incurring additional overhead.

8.2.6 Effect of Cnode/Ionode Ratio

Figure 8.15 shows the performance of the NAS benchmarks as the number of ionodes in the system increases from 1 to 16. The same cache management policy (base caching) was used in all cases.
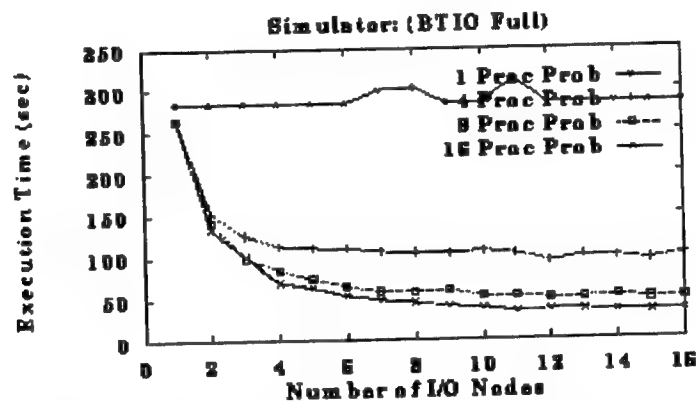
Figure 8.15: Predicted Execution Time as a Function of IOnodes and Cnodes

Since the NAS benchmarks only perform write operations, there is no variation in performance for the different management policies. The one-processor problem does not benefit from the availability of more ionodes because the I/O requests performed by this benchmark are small enough to fit inside a single data block, allowing them to be serviced by a single ionode. However, when multiple processes are used to run the benchmark, more I/O requests are generated, allowing extra ionodes to relieve the congestion which results when all target processes try to access a single ionode. It can also be seen that when a given number of processors is available in the system, performance is best when the number of cnodes and ionodes is balanced. For example, with 17 processors, performance goes from worst to best when there are: 1 cnode, 16 ionodes; 16 cnodes, 1 ionode; 4 cnodes, 13 ionodes; 9 cnodes, 8 ionodes.

8.3   Related Work

Most simulation engines use sequential or parallel implementations of the quantum protocol. Among these are Proteus (Brewer et al. 1991), a parallel architecture simulation engine, Tango (Davis et al. 1991), a shared memory architecture simulation engine, Wisconsin Wind Tunnel (Reinhardt et al. 1993), a shared memory architecture simulation engine and SimOS, a complete

system simulator (multiple programs plus operating system). Two simulation engines which use approaches similar to ours are Parallel Proteus (Legedza and Weihl 1996) and LAPSE.

Parallel Proteus is the parallelization of the Proteus simulation engine, which uses the quantum protocol. The synchronization overhead caused by frequent barriers is reduced using two methods: (a) predictive barriers and (b) local barriers. Predictive barriers is a method for safely increasing the simulation quantum beyond L, the minimum communication latency of the target machine. This method uses runtime and compile time analysis to determine, at the beginning of a simulation quantum, the earliest simulation time at which any LP will send a message to any other LP. Consequently, the simulation quantum can be extended until that time. Runtime analysis involves simply running an LP until it communicates. If it stops at the equivalent of a receive statement, analysis performed at compile time is used to predict when it would have sent a message if it were instantly resumed. The method of local barriers uses statically available communication topology information (i.e., groups of LPs that communicate only within the groups they belong to) to reduce the global synchronization at the end of a simulation quantum to local synchronizations between groups of LPs.

LAPSE (Dickens et al. 1994) is a parallel simulation engine for programs using the message passing library of the Intel Paragon. It uses a quantum protocol called WHOA (window-based halting on appointments). Like Parallel Proteus, it uses runtime analysis to determine the size of the simulation quantum, but the runtime analysis is not supplemented with compile time analysis.

In comparison, we use the equivalent of runtime analysis since we execute an LP until it reaches a receive statement. The benefits of compile time analysis are achieved using the conditional event protocol, which is portable and does not need target instruction set analysis. In addition, our implementation of the null message protocol adapts automatically to the dynamically changing communication topology specified by the target program. Perhaps most importantly, it automatically recognizes (some forms of) deterministic code and switches off all synchronization while simulating it; automatic recognition of other forms of determinism are being added to the simulator. As seen in the results, this optimization helps us eliminate almost all the synchronization overhead in simulating many real applications.

Related work in parallel I/O includes three approaches to collective I/O discussed in (Kotz 1997): traditional caching, two-phase I/O, and disk-directed I/O. Traditional caching does no collective I/O optimizations, since I/O requests are served as they arrive. These three methods were implemented and compared using the STARFISH (Kotz 1996) simulator, which is based on Proteus (Brewer et al. 1991), a parallel architecture simulation engine.

In (Baylor et al. 1994), a hybrid methodology for evaluating the performance of parallel I/O subsystems was done. PIOS, a trace-driven I/O simulator, is used to calculate the performance of the I/O system for a subset of the problem to be evaluated, while an analytical model was used for the remainder. PIOS uses a synthetically generated workload and models the Vulcan network (Baylor et al. 1996).

Panda (Seamons et al. 1995) provides a high-level collective I/O library interface. It implements server-directed I/O, which is disk-directed I/O at the logical file level, rather than the physical disk level. This method provides a high-level of portability by avoiding the difficult details of utilizing specific attributes for each underlying filesystem. Unfortunately, it may not produce as much performance as a true implementation of disk-directed I/O might. Also, Panda provides its own API to access its libraries. This may not be desirable since it is non-standard, which is one of the reasons why something such as MPI-IO has been proposed.

(Dahlin et al. 1994) presented a number of cooperative caching techniques, but only within the framework of a network environment. (Cortes et al. 1997) extended cooperative caching to include parallel machines by allowing the speed of the interconnection network to be varied. However, all their results were obtained using the Sprite network workload, which may not be very representative of typical parallel machine workloads. (Cortes et al. 1997) also introduced a cooperative caching strategy which eliminates the need for coherency mechanisms by coordinating the contents of one hundred percent of the system's file cache.

9. CONCLUSIONS

The SESAME project has produced valuable results of several different sorts:

- **Proof of concept.** The OS kernel instrumentation represents a new and better way of measuring system performance, which has now been clearly demonstrated to work.

- **Programming.** The code for the project's various programs is online in the repository; it is available both to DARPA and the research community at large.

- **Results.** The modeling team published a number of papers with valuable insight into system performance.

- **Managerial lessons.** Not all problems were technical.

9.1   Proof of Concept

Although all phases of the project required creative approaches and original programming, the kernel instrumentation was particularly innovative. The goal was to produce a portable, flexible, extensible method for recording system activity that would allow us to investigate performance inside the OS the way the Pablo project measured the application level. At the start, it was by no means clear that this could be done for enough kernel components, and without seriously disturbing system performance.

The results obtained indicate that this has been achieved. We now have generalized methods for collecting, extracting and storing data. The data format has been shown to be suitable for automated handling, and the information to be revealing. The approach appears to have so little impact on system performance that it could be fielded in OS releases, at least in beta test versions.

It should be emphasized again that while this test was done on an MPP system, most of the technique is easily portable to other environments. Distributed systems would also be targets for this approach.

9.2   Programming

All parts of the project involved original programming, and most of the code is online in the repository. The portable sections of the kernel instrumentation are there, along with all the handling and presentation routines developed by the repository staff. Code for the Paragon version of the simulator is available, along with links to releases of other software produced by the modeling team.

9.3   Results

The modeling team has published a number of papers on the IBM SP/2. Their research points the way to meaningful performance gains.

The data from the Intel Paragon instrumentation would be valuable for improving performance in the system if further development were planned. The reason this is not so, and the data is relegated to conceptual interest, is part of the other lessons learned.

9.4   Managerial Lessons

The SESAME project was extremely ambitious. The original design called for nothing less than the establishment of a research center that would develop the instrumentation, modeling, and analysis systems, and make them available to the world research community via the World Wide Web. The researchers both inside the project and participating remotely would collect data, share their analyses, and suggest changes to both the investigation software and ultimately to the operating systems themselves, correcting the deficiencies of the current generation of MPP systems.

It ran afoul of two important realities that, from a managerial standpoint, are almost as important as the techniques and results of the study.

First, the systems under investigation were proprietary commercial products. The operating system code into which the measurement hooks had to be placed, and which would have had to be modified to correct the problems found, was not available for public inspection. Intel allowed the Platinum measurement team access to the code under a non-disclosure agreement. This permitted us to take the measurements, but prevented our analysts from identifying the source of difficulties, or recommending solutions.

The Platinum team would have preferred to work on the more current IBM SP/2 computer, and expended considerable effort to obtain access to the source code. The overtures were met with polite interest, but little more.

Second, the time to produce instrumentation for such complex systems is long with respect to the product cycle. The original targets of our investigation, the Intel Paragon and the IBM SP/1, were obsolete and largely abandoned by the time we were prepared to study them. Because we were not part of early product development, and could only work on systems that were available for distribution, keeping up with the designers was inherently difficult.

The solution to these problems is for design teams and beta-testers to use the SESAME tools. The ideal test-bed program is the end-user's application, and the ideal analyst is the designer working on the problem OS module. Even used by commercial developers, SESAME techniques could be of tremendous value. An instrumented system that could be fielded to users would effectively eliminate the dreaded "can't reproduce the problem" situation, in which the user reports a problem, but the designers cannot duplicate it in the laboratory with test equipment available (one of the least tractable of difficulties).

An "open source" operating system, where the problem code is open for public inspection and tinkering, would be ideally suited to the SESAME concept. There, the full potential of bringing the means of investigation to local users and remote researchers could be fully realized.

10. REFERENCES

Bagrodia, R., S. Docy, and A. Kahn. Parallel Simulation of Parallel File Systems and I/O Programs. In *Supercomputing 97*, 1997.

Bailey, D.,T.Harris, W.Saphir, R.V.D. Wijngaart, A. Woo, and M. Yarrow. The NAS Parallel Benchmarks 2.0. Technical Report Nas-95-020, NASA Ames Research Center, Moffet Field, CA 94035-1000, December 1995.

Baylor, S.J., C. Benveniste, and L.J. Beolhouwer. A Methodology for Evaluating Parallel I/O Performance for Massively Parallel Processors. *In Proceedings of the 27th Annual Simulation Symposium*, pages 31-40, April 1994.

Brewer, E. A., C. N. Dellarocas, A. Colbrook, and W. E. Weihl. Technical Report MIT/LCS/TR-516, Massachusetts Institute of Technology, Cambridge, MA 02139, 1991.

Covington, R. G., S. Dwarkadas, J. R. Jump, J.B. Sinclair, and S. Madala. The Efficient Simulation of Parallel Computer Systems. *IJCS*, 1:31-58, 1991.

Chandy, K. M., and J. Misra. Distributed Simulation: A Case Study in Design and Verification of Distributed Programs. *IEEE Transactions on Software Engineering*, Pages 440-452, September 1979.

Chandy, K. M., and R. Sherman. The Conditional Event Approach to Distributed Simulation. In *Proceedings of the SCS Multiconference on Distributed Simulation*, Miami, Pages 93-99, 1989.

Corbett, P.F., D.G. Feitelson. The Vesta Parallel File System. *ACM Transactions on Computer Systems*, 14(3):225-264, August 1996.

Corbett, P.F., D.G. Feitelson, J-P Prost, G.S. Almasi, S.J.Baylor, A.S. Bolmarich, Y. Hsu, J. Satran, M. Snir, R. Colao, B. Herr, J. Kavaky, T.R. Morgan, and A. Zlotek. Parallel File Systems for IBM SP Computers. *IBM Systems Journal*, 34(2):222-248, January 1995.

Cotes, T., S. Girona, and J. Labarta. Avoiding the Cache-Coherence Problem in Parallel/Distributed File System. In *Proceedings of the High-Performance Computing and Networking*. Pages: 860-869, April 1997.

Dahlin , M.D., R.Y. Wang, T.E. Anderson and D.A. Patterson. Cooperative Caching: Using Remote Client Memory to Improve File System Performance. *In Proceedings of the 1994 Symposium on Operating Systems Design and Implementation*, pages 61-74, November 1994.

Davis, H., S. R. Goldschmidt, and Hennessey. Multiprocessor Simulation and Tracing Using Tango. In *Proceedings of ICPP '91*, Pages 99-107, August 1991.

Dickens, P., P. Heidelberger, and D. Nicol. A Distributed Memory Lapse: Parallel Simulation of Message-Passing Programs. In *Workshop on Parallel and Distributed Simulation*, Pages 32-38, July 1994.

Dickens, P. M., P. Heidelberger, and D.M. Nicol. Parallelized Direct Execution Simulation of Message-Passing Parallel Programs. *IEEE Transactions on Parallel and Distributed Systems*, 6(4):297-320, October 1996.

Rosario, J.M. del, R. Bordawekar, and A. Chouldhary. Improved Parallel I/O via a Two-Phase Run-Time Access Strategy. In Proceedings of the IPPS'93 Workshop on Input/Output in Parallel Computer Systems, pages 56-70, Newport Beach, CA, 1993.

Feldman, S. I., D. M. Gay, Mark W. Maimone, and N. L. Schryer. A Fortran-To-C Converter. Technical Report No. 149, AT&T Bell Laboratories, Murray Hill, NJ, May 1990.

Fineberg, S., P. Wong, B. Nitzberg, and C. Kuszmaul. PMPIO- A Portable Implementation of MPI-IO. In *Proceedings of the Sixth Symposium on the Frontiers of Massively Parallel Computation*, pages 188-195. October 1996.

Fujimoto, R. Parallel Discrete Event Simulation. *Communications of The ACM*, 33(10):30-53, October 1990.

Jha, V., and R. Bagrodia. Transparent Implementation of Conservative Algorithms In Parallel Simulation Languages. In *Winter Simulation Conference*, December 1993.

Kotz, D. Tuning STARFISH. Technical Report PCS-TR96-296, Dept. of Computer Science, Dartmouth College, October 1996.

Kotz, D. Disk-Directed I/O for MIMD Multiprocessors. ACM Transactions on Computer Systems, 15(1):41-74, February 1997.

Legedza, U., and W. E. Weihl. Reducing Synchronization Overhead in Parallel Simulation. In *Tenth Workshop on Parallel and Distributed Simulation PADS 96*, May 1996.

Misra, J. Distributed Discrete-Event Simulation. *ACM Computing Surveys*, 18(1):39-65, March 1986.

MPI Forum. MPI: A Message Passing Interface. In *Proceedings of 1993 Supercomputing Conference*, Portland, Washington, November 1993.

The MPI-IO Committee. MPI-IO: a Parallel File I/O Interface for MPI. April 1996. http://lovelace.nasa.nasa.gov/MPI-IO/mpi-io-report.0.5.ps

Nitzberg, B. Performance of iPSC/860 Concurrent File System. Technical Report RND-92-020, NAS Systems Division, NASA Ames, December 1992.

Nieuwejaar, N. and D. Kotz. The Galley Parallel File System. *In Proceedings of the 10th ACM International Conference on Supercomputing*, pages 374-381, Philadelphia, May 1996.

Prakash, S. Performance Prediction of Parallel Programs. Ph.D. Thesis, Computer Science Department, UCLA, Los Angeles, CA 90095, November 1996.

Prakash, S. and R. Bagrodia. Using Parallel Simulation to Evaluate MPI Programs. *Proceedings of the 1998 Winter Simulation Conference - WSC '98.*

Reinhardt, S. K., M. D. Hill, J. R. Larus, A. R. Lebeck, J. C. Lewis, and D. A. Wood. The Wisconsin Wind Tunnel: Virtual Prototyping of Parallel Computers. In *Proceedings of the 1993 ACM Sigmetrics Conference*, May 1993.

Rosenblum, M. E. Begnion, S. Devine, and S. A. Herrod. Using The SimOS Machine Simulator to Study Complex Computer Systems. *ACM Transactions on Modeling and Computer Simulation*, 7(1), January 1997.

Rosenblum, M. S. A. Herrod, E. Witchel, and A. Gupta. Complete Computer System Simulation: The SimOS Approach. *IEEE Parallel and Distributed Technology*. Vol. 3, No. 4, Winter 1995.

Seamons, K.E., Y. Chen, P. Jones, J. Jozwiak, and M. Winslett. Server-directed Collective I/O in Panda. In *Proceedings of Supercomputing '95*, San Diego. 1995.

Sanders, D., Y. Park, and M. Brodowicz. Implementation and Performance of MPI-IO File Access using MPI datatypes. Technical Report UH-CS-96-12, University of Houston, November 1996.

DISTRIBUTION LIST

addresses	number of copies
DR. RAYMOND A. LIUZZI AFRL/IFTD 525 BROOKS ROAD ROME, NY 13441-4505	10
UNIV OF CALIFORNIA,LOS ANGELES COMPUTER SCIENCE DEPARTMENT 1400 PETER UEBERROTH BLDG. 405 HILGARD AVENUE LOS ANGELES, CA 90024-1406	5
AFRL/IFOIL TECHNICAL LIBRARY 26 ELECTRONIC PKY ROME NY 13441-4514	1
ATTENTION: DTIC-OCC DEFENSE TECHNICAL INFO CENTER 8725 JOHN J. KINGMAN ROAD, STE 0944 FT. BELVOIR, VA 22060-6218	1
DEFENSE ADVANCED RESEARCH PROJECTS AGENCY 3701 NORTH FAIRFAX DRIVE ARLINGTON VA 22203-1714	1
ATTN: NAN PFRIMMER IIT RESEARCH INSTITUTE 201 MILL ST. ROME, NY 13440	1
AFIT ACADEMIC LIBRARY AFIT/LDR, 2950 P.STREET AREA B, BLDG 642 WRIGHT-PATTERSON AFB OH 45433-7765	1
AFRL/MLME 2977 P STREET, STE 6 WRIGHT-PATTERSON AFB OH 45433-7739	1

```
USAF/AIR FORCE RESEARCH LABORATORY              1
AFRL/VSOSA(LIBRARY-BLDG 1103)
5 WRIGHT DRIVE
HANSCOM AFB   MA   01731-3004


ATTN:   EILEEN LADUKE/D460                      1
MITRE CORPORATION
202 BURLINGTON RD
BEDFORD MA 01730


OUSD(P)/DTSA/DUTD                               1
ATTN:   PATRICK G. SULLIVAN, JR.
400 ARMY NAVY DRIVE
SUITE 300
ARLINGTON VA 22202

SOFTWARE ENGR'G INST TECH LIBRARY              1
ATTN:   MR DENNIS SMITH
CARNEGIE MELLON UNIVERSITY
PITTSBURGH PA 15213-3890


USC-ISI                                         1
ATTN:   DR ROBERT M. BALZER
4676 ADMIRALTY WAY
MARINA DEL REY CA 90292-6695


KESTREL INSTITUTE                               1
ATTN:   DR CORDELL GREEN
1801 PAGE MILL ROAD
PALO ALTO CA 94304


ROCHESTER INSTITUTE OF TECHNOLOGY              1
ATTN:   PROF J. A. LASKY
1 LOMB MEMORIAL DRIVE
P.O. BOX 9887
ROCHESTER NY 14613-5700

AFIT/ENG                                        1
ATTN:TOM HARTRUM
WPAFB OH 45433-6583



THE MITRE CORPORATION                           1
ATTN:   MR EDWARD H. BENSLEY
BURLINGTON RD/MAIL STOP A350
BEDFORD MA 01730
```

ANDREW A. CHIEN   1
SAIC CHAIR PROF (SCI APL INT CORP)
USCD/CSE-AP&M 4808
9500 GILMAN DRIVE, DEPT. 0114
LAJOLLA CA 92093-0114

                                                   1
HONEYWELL, INC.
ATTN: MR BERT HARRIS
FEDERAL SYSTEMS
7900 WESTPARK DRIVE
MCLEAN VA 22102

                                                   1
SOFTWARE ENGINEERING INSTITUTE
ATTN: MR. WILLIAM E. HEFLEY
CARNEGIE-MELLON UNIVERSITY
304 OAK GROVE CT
WESFORD PA 15090

                                                   1
UNIVERSITY OF SOUTHERN CALIFORNIA
ATTN: DR. YIGAL ARENS
INFORMATION SCIENCES INSTITUTE
4676 ADMIRALTY WAY/SUITE 1001
MARINA DEL REY CA 90292-6695

                                                   1
COLUMBIA UNIV/DEPT COMPUTER SCIENCE
ATTN: DR GAIL E. KAISER
450 COMPUTER SCIENCE BLDG
500 WEST 120TH STREET
NEW YORK NY 10027

                                                   1
AFIT/ENG
ATTN: DR GARY B. LAMONT
SCHOOL OF ENGINEERING
DEPT ELECTRICAL & COMPUTER ENGRG
WPAFB OH 45433-6583

                                                   1
NSA/OFC OF RESEARCH
ATTN: MS MARY ANNE OVERMAN
9800 SAVAGE ROAD
FT GEORGE G. MEADE MD 20755-6000

                                                   1
TEXAS INSTRUMENTS INCORPORATED
ATTN: DR DAVID L. WELLS
P.O. BOX 655474, MS 238
DALLAS TX 75265

                                                   1
KESTREL DEVELOPMENT CORPORATION
ATTN: DR RICHARD JULLIG
3260 HILLVIEW AVENUE
PALO ALTO CA 94304

```
DARPA/ITO                                            1
ATTN:  DR KIRSTIE BELLMAN
3701 N FAIRFAX DRIVE
ARLINGTON VA 22203-1714


NASA/JOHNSON SPACE CENTER                            1
ATTN:  CHRIS CULBERT
MAIL CODE PT4
HOUSTON TX 77058


STERLING IMD INC.                                    1
KSC OPERATIONS
ATTN:  MARK MAGINN
BEECHES TECHNICAL CAMPUS/RT 26 N.
ROME NY 13440

SCHLUMBERGER LABORATORY FOR                          1
   COMPUTER SCIENCE
ATTN:  DR. GUILLERMO ARANGO
8311 NORTH FM620
AUSTIN, TX 78720

DECISION SYSTEMS DEPARTMENT                          1
ATTN:  PROF WALT SCACCHI
SCHOOL OF BUSINESS
UNIVERSITY OF SOUTHERN CALIFORNIA
LOS ANGELES, CA 90039-1421

NATIONAL INSTITUTE OF STANDARDS                      1
   AND TECHNOLOGY
ATTN:  CHRIS DABROWSKI
ROOM A266, BLDG 225
GAITHSBURG MD 20899


EXPERT SYSTEMS LABORATORY                            1
ATTN:  STEVEN H. SCHWARTZ
NYNEX SCIENCE & TECHNOLOGY
500 WESTCHESTER AVENUE
WHITE PLAINS NY 20604


NAVAL TRAINING SYSTEMS CENTER                        1
ATTN:  ROBERT BREAUX/CODE 252
12350 RESEARCH PARKWAY
ORLANDO FL 32826-3224


DR JOHN SALASIN                                      1
DARPA/ITO
3701 NORTH FAIRFAX DRIVE
ARLINGTON VA 22203-1714
```

DR BARRY BOEHM   1
DIR, USC CENTER FOR SW ENGINEERING
COMPUTER SCIENCE DEPT
UNIV OF SOUTHERN CALIFORNIA
LOS ANGELES CA 90089-0781

DR STEVE CROSS   1
CARNEGIE MELLON UNIVERSITY
SCHOOL OF COMPUTER SCIENCE
PITTSBURGH PA 15213-3891

DR MARK MAYBURY   1
MITRE CORPORATION
ADVANCED INFO SYS TECH; G041
BURLINTON ROAD, M/S K-329
BEDFORD MA 01730

ISX   1
ATTN: MR. SCOTT FOUSE
4353 PARK TERRACE DRIVE
WESTLAKE VILLAGE,CA 91361

MR GARY EDWARDS   1
ISX
433 PARK TERRACE DRIVE
WESTLAKE VILLAGE CA 91361

LEE ERMAN   1
CIMFLEX TEKNOWLEDGE
1810 EMBACADERO ROAD
P.O. BOX 10119
PALO ALTO CA 94303

DR. DAVE GUNNING   1
DARPA/ISO
3701 NORTH FAIRFAX DRIVE
ARLINGTON VA 22203-1714

DR. MICHAEL PITTARELLI   1
COMPUTER SCIENCE DEPART
SUNY INST OF TECH AT UTICA/ROME
P.O. BOX 3050
UTICA, NY 13504-3050

CAPRARO TECHNOLOGIES, INC   1
ATTN: GERARD CAPRARO
311 TURNER ST.
UTICA, NY 13501

DL-6

USC/ISI   1
ATTN: BOB MCGREGOR
4676 ADMIRALTY WAY
MARINA DEL REY, CA 90292

SRI INTERNATIONAL   1
ATTN: ENRIQUE RUSPINI
333 RAVENSWOOD AVE
MENLO PARK, CA 94025

DARTMOUTH COLLEGE   1
ATTN: DANIELA RUS
DEPT OF COMPUTER SCIENCE
11 ROPE FERRY ROAD
HANOVER, NH 03755-3510

UNIVERSITY OF FLORIDA   1
ATTN: ERIC HANSON
CISE DEPT 456 CSE
GAINESVILLE, FL 32611-6120

CARNEGIE MELLON UNIVERSITY   1
ATTN: TOM MITCHELL
COMPUTER SCIENCE DEPARTMENT
PITTSBURGH, PA 15213-3890

UNIVERSITY OF ROCHESTER   1
ATTN: JAMES ALLEN
DEPARTMENT OF COMPUTER SCIENCE
ROCHESTER, NY 14627

MNIS-TEXTWISE LABS   1
ATTN: PARAIE SHERIDAN
DEY CENTENNIAL PLAZA 5TH FLOOR
SYRACUSE, NY 13502

WRIGHT STATE UNIVERSITY   1
ATTN: DR. BRUCE BERPA
DEPART OF COMPUTER SCIENCE & ENGIN
DAYTON, OHIO 45435-0001

UNIVERSITY OF FLORIDA   1
ATTN: SHARMA CHAKRAVARTHY
COMPUTER & INFOR SCIENCE DEPART
GAINESVILLE, FL 32622-6125

KESTREL INSTITUTE
ATTN: DAVID ESPINOSA
3260 HILLVIEW AVENUE
PALO ALTO, CA 94304

1

USC/INFORMATION SCIENCE INSTITUTE
ATTN: DR. CARL KESSELMAN
11474 ADMIPALTY WAY, SUITE 1001
MARINA DEL REY, CA 90292

1

MASSACHUSETTS INSTITUTE OF TECH
ATTN: DR. MICHAELE SIEGEL
SLOAN SCHOOL
77 MASSACHUSETTS AVENUE
CAMBRIDGE, MA 02139

1

USC/INFORMATION SCIENCE INSTITUTE
ATTN: DR. WILLIAM SWARTHOUT
11474 ADMIRALTY WAY, SUITE 1001
MARINA DEL REY, CA 90292

1

STANFORD UNIVERSITY
ATTN: DP. GIO WIEDERHOLD
857 SIERRA STREET
STANFOPD
SANTA CLARA COUNTY, CA 94305-4125

1

SPAWARSYSCEN D44209
ATTN: LEAH WONG
53245 PATTERSON ROAD
SAN DIEGO, CA 92152-7151

1

SPAWARSYSCEN D4123
ATTN: LES ANDERSON
53560 HULL STREET
SAN DIEGO CA 92152-5001

1

GEORGE MASON UNIVERSITY
ATTN: SUSHIL JAJODIA
ISSE DEPT
FAIRFAX, VA 22030-4444

1

DIRNSA
ATTN: MICHAEL R. WARE
DOD, NSA/CSS (R23)
FT. GEORGE G. MEADE MD 20755-6000

1

DR. JIM RICHARDSON
3660 TECHNOLOGY DRIVE
MINNEAPOLIS, MN 55418

1

1

LOUISIANA STATE UNIVERSITY
COMPUTER SCIENCE DEPT
ATTN: DR. PETER CHEN
257 COATES HALL
BATON ROUGE, LA 70803

1

INSTITUTE OF TECH DEPT OF COMP SCI
ATTN: DR. JAIDEEP SRIVASTAVA
4-192 EE/CS
200 UNION ST SE
MINNEAPOLIS, MN 55455

1

GTE/BBN
ATTN: MAURICE M. MCNEIL
9655 GRANITE RIDGE DRIVE
SUITE 245
SAN DIEGO, CA 92123

1

UNIVERSITY OF FLORIDA
ATTN: DR. SHARMA CHAKRAVARTHY
E470 CSE BUILDING
GAINESVILLE, FL 32611-6125

1

AFRL/IFT
525 BROOKS ROAD
ROME, NY 13441-4505

1

AFRL/IFTM
525 BROOKS ROAD
ROME, NY 13441-4505

1

JEAN SCHOLTZ
DARPA/ITO
3701 NORTH FAIRFAX DRIVE
ARLINGTON VA 22203-1714

1

DR. ROGER CHEN
DEPT OF ELECT & COMPUTER ENGINR
SYRACUSE UNIVERSITY
SYRACUSE, NY 13244-1240

MISSION
OF
AFRL/INFORMATION DIRECTORATE (IF)

The advancement and application of information systems science and technology for aerospace command and control and its transition to air, space, and ground systems to meet customer needs in the areas of Global Awareness, Dynamic Planning and Execution, and Global Information Exchange is the focus of this AFRL organization. The directorate's areas of investigation include a broad spectrum of information and fusion, communication, collaborative environment and modeling and simulation, defensive information warfare, and intelligent information systems technologies.